Vysoká škola báňská – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Dynamické lineární hašování zachovávající uspořádání

Dynamic Linear Hashing with Order Preserving

2015 **Petr Tureček**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Petr Tureček**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Dynamické lineární hašovaní zachovávající uspořádání
Dynamic Linear Hashing with Order Preserving

Zásady pro vypracování:

Hashovací tabulka je datová struktura využívaná zejména v případech, kdy je požadováno dotazování s konstantní složitostí. Struktura využívá hašovací funkci pro transformaci klíče na index, podle kterého přistupuje k jednotlivých záznamům. Hlavními dvěma problémy hashovací tabulky je nutnost znát dopředu objem vkládaných dat a neschopnost vykonávání rozsahových dotazů. Existující modifikace se snaží zmiňované problémy vyřešit pomocí dynamického hashování uchovávající uspořádání. Cílem této práce je implementace a testování různých modifikací hashovací tabulky.

1. Nastudujte varianty podporující dynamické hashování (zejména Litwinovo lineární a Faginovo rozšiřitelné hashování).
2. Nastudujte varianty podporující uspořádání klíčů.
3. Naimplementujte nastudované přístupy do stávajícího frameworku QuickDB.
4. Porovnejte výkon naimplementovaných přístupů s výkonem ostatních datových struktur.
5. Výsledky experimentů vyhodnoťte.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Peter Chovanec**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: *29. července 2015*

podpis studenta

Poděkování

Rád bych poděkoval Ing. Peteru Chovancovi za odbornou pomoc a konzultaci při vytváření této bakalářské práce.

Abstrakt

Hašovací tabulka je hojně používaná datová struktura, ceněná pro rychlý zápis a čtení dat. Data jsou však interně uložena v náhodném pořadí a není možné vyhledat uložené klíče v určitém rozsahu, aniž bychom museli projít všechny uložené záznamy. Cílem bakalářské práce je implementace tří algoritmů hašování, které budou doplněny o schopnost zachovávání pořadí, tedy že záznamy vložené do vytvořených hašovacích tabulek budou uloženy seřazené dle svých klíčů. Taková hašovací tabulka by mohla konkurovat stromům v nasazení v databázových systémech. Implementovány budou jmenovitě lineární hašování, rozšířitelné hašování a hašovací tabulka s pevnou velikostí. Tyto algoritmy jsou již mnoho let známy a běžně používány. Projekt bude vyvíjen v prostředí Microsoft Visual Studio 2013, v programovacím jazyce C++. Třídy hašovacích tabulek budou začleněny do frameworku RadegastDB (dříve QuickDB), který je vyvíjen na Katedře informatiky VŠB-TU Ostrava.

Klíčová slova

Hašování; Hašovací tabulka; Dynamické hašování; Lineární hašování; Rozšířitelné hašování; Hašování zachovávající pořadí

Abstract

Hash table is widely used data structure, valued for its fast writing and reading of data. But the data are stored in random like order internally and it is not possible to perform range queries over the hash table without checking all the stored records. The bachelor thesis goal is to implement three hashing algorithms, all altered to provide preserving the order, thus the items inserted into hash tables will be stored in a sorted order according to the values of their keys. Such hash tables could be competitive to trees in the deployment in database systems. Implemented will be namely the linear and the extendible hashing, and the hash table with fixed size. These algorithms are well known for many years and commonly used. The project will be developed in Microsoft Visual Studio 2013 and the C++ programming language will be used. Classes of hash tables will be integrated into the RadegastDB framework (formerly QuickDB), which is being developed at Computer Science Department of VŠB-TU Ostrava.

Key words

Hashing; Hash table; Dynamic hash table; Linear hashing; Extendible hashing; Order preserving hash table

Seznam použitých termínů

Termín	Anglicky	Význam termínu
Hašovací tabulka	Hash table	Vyhledávací datová struktura pro ukládání záznamů typu uspořádaná dvojice [klíč]:[hodnota], která pomocí hašovací funkce transformuje hodnotu klíče do adresního prostoru datové struktury.
Hašovací funkce	Hash function	Hašovací funkce je matematická funkce (resp. algoritmus) pro převod vstupních dat do (relativně) malého čísla.
Otisk	Hash value	Výstup hašovací funkce používaný jako index pro adresaci v hašovací tabulce.
Faktor naplnění	Load factor	Poměr obsazených uzlů k velikosti hašovací tabulky.
Kolize	Collision	Kolize je situace, kdy se záznamy s různými klíči hašují zvolenou funkcí na stejné místo.
Uzel	Bucket	Uzel je seznam prvků zahašovaných (majících stejnou hash value) na stejnou pozici v hašovací tabulce. Může být realizován různými datovými strukturami - stromem, zřetězeným seznamem, setřízeným seznamem i další hašovací tabulkou. *Pozn.: Anglický výraz bucket se nejčastěji překládá jako vědro, nebo kyblík, užití se liší autor od autora a jeden ustálený výraz česká IT terminologie zatím nemá. Nadále se budu držet pojmu uzel (angl. node), který se používá u stromů, mimo jiné i proto že v rámci frameworku, ve kterém bude implementace probíhat, jsou tyto uzly pro uložení záznamů realizovány třídou cNode původně napsanou pro stromy.*
Pole	Array	Datová struktura, která sdružuje daný, vždy konečný počet prvků (čísel, textových řetězců, …) stejného datového typu. K jednotlivým prvkům pole se přistupuje pomocí jejich indexu (celého čísla, označujícího pořadí prvku).
Strom	Tree	Široce využívaná datová struktura, která představuje stromovou strukturu s propojenými uzly. Uzly jsou navzájem spojeny „hranami“.

Lineární seznam	Linked list	Datová struktura, vzdáleně podobná poli (umožňuje uchovat velké množství hodnot ale jiným způsobem), obsahující jednu a více datových položek (struktur) stejného typu, které jsou navzájem lineárně provázány vzájemnými odkazy pomocí ukazatelů nebo referencí.
Seřazený seznam	Sorted list	Datová struktura uchovávající záznamy seřazené podle hodnoty. Seřazení záznamů umožňuje rychlejší vyhledávání, nevýhodou je pomalejší vkládání, během kterého musí být vkládaný prvek zařazen na správnou pozici.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Cílem této bakalářské práce je implementace dvou algoritmů dynamického hašování, jmenovitě lineárního hašování a rozšířitelného (extendible) hašování, a hašovací tabulky s pevnou velikostí, upravených tak aby výsledný algoritmus zachovával pořadí, tedy aby záznamy vložené do vytvořených hašovacích tabulek byly v datové struktuře uloženy seřazené dle svých klíčů. Algoritmy budou implementovány ve vývojovém prostředí Microsoft Visual Studio 2013, v programovacím jazyce C++, v rámci frameworku RadegastDB (dříve QuickDB), který je vyvíjen na Katedře informatiky, Fakulty elektrotechniky a informatiky, Vysoké školy báňské – Technické univerzity Ostrava.

V první části práce rozeberu obecnou teorii hašování, základní pojmy a principy. Druhá kapitola se zaměří na popis dynamického hašování, tedy hašovacích tabulek schopných měnit za běhu svou velikost. Detailně se pak budu věnovat v rámci této práce implementovaným algoritmům lineárního a rozšířitelného hašování, v podobě v jaké byly publikovány jejich autory, zatímco mnou provedené změny algoritmů nutné k dosažení požadované vlastnosti zachovávání pořadí a další specifika této konkrétní implementace výše zmíněných datových struktur budou podrobně rozebrány v kapitole třetí. Čtvrtá a poslední část této práce se zaměří na změření a podrobnou analýzu výkonnostních parametrů a vzájemné porovnání vytvořených algoritmů mezi sebou a s vybranými konkurenčními datovými strukturami (R-strom a binární strom) a zhodnocení vhodnosti použití takto upravených hašovacích tabulek v databázových systémech.

1 Stručný úvod do hašování

1.1 Hašovací tabulka

Hašovací tabulka (hash table, hashovací tabulka) je slovníková struktura velkého praktického významu a může být velmi efektivní. Záznamy typu klíč:hodnota ukládá pomocí hašovací funkce h(), která transformuje hodnotu klíče do adresního prostoru datové struktury [1]. Jako příklad můžeme uvést množinu záznamů, ke kterým budeme přistupovat pomocí klíče reprezentovaného řetězcem znaků, např. jménem osoby. Běžné *pole* záznamů (array) známé z programovacích jazyků umožňuje indexaci pouze ordinálními datovými typy, většinou se jedná o int (Integer). Adresace řetězcem není možná. V takovém případě se zavolá hašovací funkce, jejímž parametrem bude příslušný klíč a hašovací funkce z jeho obsahu vypočte *otisk* klíče (hash value), celočíselnou hodnotu kterou již lze použít k adresaci klasického datového pole záznamů. Využití se tedy nabízí pro rychlé vyhledávání položky v poli nebo v jiném homogenním datovém typu podle klíčů, které nejsou celočíselnou hodnotou (např. text, číslo s plovoucí desetinnou čárkou, datum), nebo klíčů, které jsou sice celočíselné, ale netvoří souvislé řady hodnot a při jejich použití k indexaci pole by v datové struktuře byly velké nevyužité bloky paměti. Takovým případem je např. rodné číslo, nebo telefonní číslo. Jedná se o celočíselné hodnoty, ale vytvořit tabulku indexovanou devítimístným telefonním číslem (=1 miliarda záznamů) pro uložení dvou tisíc jmen v telefonním seznamu je značně neefektivní. Kombinací klasické datové tabulky a hašovací funkce jsme vytvořili hašovací tabulku.

Obvykle požadujeme, aby výpočet hašovací funkce byl rychlý, vypočítané klíče byly náhodné a rovnoměrně rozdělené mezi indexy tabulky a aby podobné hodnoty měly vzdálené klíče. Za těchto podmínek můžeme rovnoměrně po celé ploše tabulky uložit i hodnoty s velmi podobnými klíči. Daní za to pak ale je nemožnost vykonávat nad takovou datovou strukturou rozsahové dotazy (range query). Popisované vlastnosti hašovací funkce nám pomohou vytvořit datovou strukturu, která efektivně využívá paměť a je rychlá. Při správně zvolené velikosti (počtu položek) homogenní datové struktury a vhodně zvolené hašovací funkci má tento algoritmus složitost v průměrném (tj. očekávaném) případě shora omezenou na O(1) – viz tabulka 1. Je tedy paměťově efektivnější než pole záznamů a umožňuje indexaci různorodými klíči. Zároveň je rychlejší než alternativní datové struktury pro ukládání záznamů typu klíč:hodnota jako jsou *lineární seznam* (linked list), *seřazený seznam* s vyhledáváním půlením intervalů (sorted list), ale i *stromy* (tree), neboť tyto metody musí při vyhledávání procházet a porovnávat více (i desítky) záznamů, zatímco hašování v ideálním případě nalezne uložený prvek na první pokus.

Tabulka 1 *Složitost operací datových struktur*

Datová struktura	Lineární seznam	Řazený seznam	Binární strom	Hašovací tabulka
Hledání	O(n)	O(log n)	O(log n)	O(1)
Vkládání	O(1)	O(n)	O(log n)	O(1)
Vyjmutí	O(n)	O(n)	O(log n)	O(1)

V předchozím odstavci bylo uvedeno, že pro dosažení optimálních výsledků při práci s hašovací tabulkou záleží na správně zvolené velikosti tabulky. Definujeme proto *faktor naplnění* (load factor) jako poměr počtu obsazených uzlů k velikosti hašovací tabulky. U základního pojetí hašovací tabulky, kdy jednotlivými položkami tabulky jsou přímo ukládané datové záznamy a na jednu adresu v hašovací tabulce (uzel) lze uložit pouze jeden záznam může faktor naplnění nabývat hodnot od 0 do 1 (včetně). Více záznamů než je velikost tabulky uložit nelze. S faktorem naplnění blížícím se 1 přibývá kolizí hašování.

1.2 Kolize

Kolize (collision) je situace, kdy se záznamy s různými klíči adresují zvolenou hašovací funkcí na stejné místo. Protože potenciálních klíčů je typicky více, než je velikost tabulky, jsou kolize, pokud předem neznáme množinu vkládaných prvků, prakticky nevyhnutelné [1]. Na rozdíl od indexování v klasickém poli, při hašování možnost vzniku kolizí vyžaduje, abychom v tabulce navíc testovali, zdali je na místě určeném hašovací funkcí prvek s hledaným klíčem, nebo prvek jiný, kterému algoritmus hašování pouze přiřadil stejnou adresu.

Podle způsobu řešení kolizí se hašovací tabulky dělí na dva základní druhy, a to:

- *Otevřená adresace* (open addressing), nebo
- *Řešení kolizí zřetězením* (separate chaining)

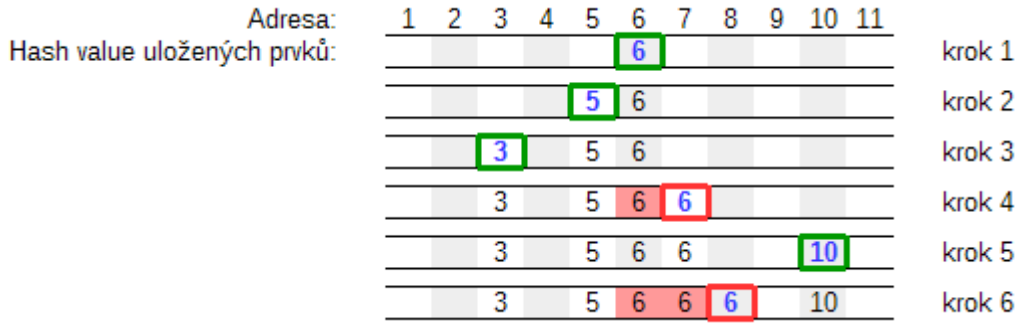
Za speciální varianty pak lze považovat situace, kdy:

- Kolize nevznikají (tzv. perfektní hašování, pro předem známou množinu klíčů),
- Kolize se ignorují a řeší prostým přepisováním (např. v transpozičních tabulkách).

V posledním případě není zaručeno, že data vložená do tabulky po čase najdeme tamtéž, proto toto řešení není vhodné pro klasickou hašovací tabulku užívanou jako datové úložiště. Tuto strukturu lze použít například jako softwarovou cache, která nám poskytne rychlý přístup k nedávno přistupovaným položkám, ale pokud hledanou položku v cache nenalezneme (byla přepsána), máme ji spolehlivě uloženou nebo dostupnou na jiném místě.

1.2.1 Otevřená adresace

Otevřená adresace je základní varianta hašovacích tabulek a je obvykle vyučována pro svou jednoduchost, názornost a možnost implementace bez použití pointerů. Tabulka má záznamy uloženy přímo ve svých položkách. Kolize adresace se řeší tak, že pokud je při vkládání pozice již obsazena, je nějakým algoritmem určeno náhradní místo, případně opakovaně, dokud se nepodaří vkládaný prvek umístit na volné místo. Při vyhledávání je nutno projít stejnou posloupnost míst, dokud není prvek nalezen anebo se nenarazí na volné místo, což znamená, že hledaný prvek v tabulce není. Protože při otevřené adresaci jsou všechny prvky přímo v tabulce, je faktor naplnění nutně menší než 1. Obrázek 1.1 ilustruje postupné ukládání prvků v odpovídající části tabulky po vložení skupiny záznamů, jejichž klíče mají přiřazené otisky (hash value) 6, 5, 3, 6, 10, 6. U klíčů s otiskem 6 dochází v krocích 4 a 6 ke kolizi a záznamy jsou vloženy na nejbližší volnou pozici.

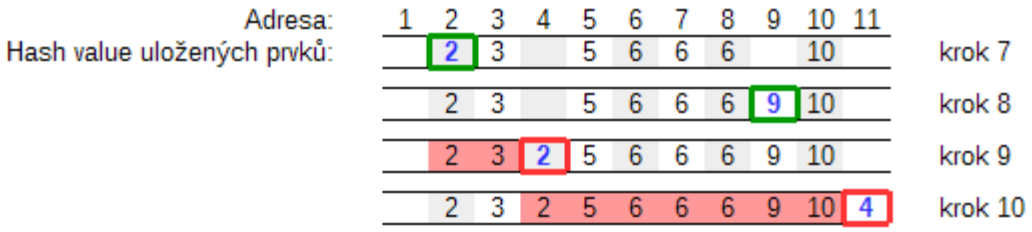


Obrázek 1.1 *Otevřená adresace - vkládání*

Metodou používanou pro určení náhradních míst je např. *lineární zkoušení* (linear probing) - hledání prvního volného záznamu za hašovací adresou. Při hledání je pak nutné projít postupně všechny záznamy mezi hašovací adresou a skutečnou pozicí uloženého záznamu. V případě že se na jednom místě nahromadí více záznamů (tento jev je v angličtině nazýván clustering), jsou tímto shlukem ovlivněny operace nad všemi záznamy adresovanými uvnitř a bezprostředně před tuto koncentrovanou oblastí. Jak nám názorně ukazuje obrázek 1.2 po vložení dalších záznamů s hash value 2, 9, 2, 4 je poslední záznam vložen až 7 buněk za adresu, která mu přísluší, i když je jediným vloženým prvkem s otiskem 4. Stejných 7 buněk pak bude muset algoritmus projít a zkontrolovat klíče vložených záznamů, než najde požadovaný prvek, který měl být na pozici 4. Zde jasně vidíme, proč od hašovací funkce požadujeme rovnoměrné rozprostření hash value po celé tabulce a umístění podobných klíčů daleko od sebe. Toto hromadění záznamů lze zmírnit tím, že se volné místo nebude hledat na pozici H+1, ale zvolí se větší krok (s oblibou se používají prvočísla) a druhý prvek se uloží třeba na pozici H+17, třetí na H+34 atd.

Další možností je *druhotné hašování* (double hashing), kdy se v případě kolize použije na klíč alternativní hašovací funkce, od které očekáváme, že bude prvek adresovat na jinou, dosud neobsazenou pozici. Podle zvolené metody a různorodosti klíčů se tabulky naplňují nejvýše na 70-90% (faktor naplnění 0,7-0,9), protože pak už se doby operací výrazně prodlužují. Při naplnění je nutno přehašovat do nové, větší tabulky.

Mezi další nevýhody otevřené adresace pak patří složité mazání a přehašování prvků. Protože jsou kolize řešeny sekvenčním ukládáním prvků se stejnou hašovací adresou za sebe, znamenalo by prosté smazání jednoho prvku uprostřed této posloupnosti ztrátu přístupu ke

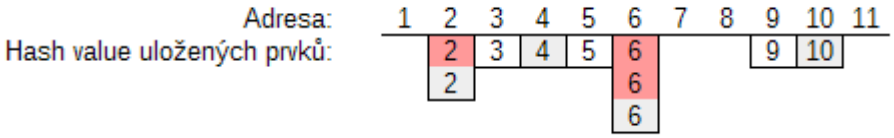


Obrázek 1.2 *Otevřená adresace - kolize*

zbývajícím prvkům, protože prázdná pozice v hašovací tabulce indikuje, že hledání skončilo. Pokud se vrátíme k předchozímu obrázku, po odstranění prvku na pozici 9 by algoritmus při hledání záznamu s hodnotou H=4 musel zkontrolovat 5 nesouvisejících záznamů a ten správný by vůbec nenašel, protože by se zastavil na prázdné pozici 9. Záznamy za odstraněným prvkem se tak musí znovu přehašovat, včetně těch, které jsou na svém místě (zde záznam na pozici 10) a zřetězené záznamy se přesunou. Lepším řešením se zdá být označení pozice odstraněného prvku speciální hodnotou vyjadřující, že tento prvek je neplatný a hledání má pokračovat dále. Tato zneplatněná pozice pak může být využita nově vloženým prvkem, případně se tabulka přehašuje později, až bude neplatných prvků větší množství. Hromadění neplatných prvků v tabulce má za následek prodloužení operací s hašovací tabulkou (prvky je nutno při hledání přeskakovat).

1.2.2 **Zřetězení záznamů**

Při řešení kolizí zřetězením záznamů každá položka tabulky obsahuje ukazatel na *uzel* - seznam prvků zahašovaných na stejné místo (bucket). Samotný uzel může být realizován různými způsoby – *strom*, *seřazený seznam* nebo *lineární seznam*. Při hledání pak už jen projdeme seznam prvků hašovaných na stejnou adresu, což bývá u hašovacích tabulek s faktorem naplnění blížícím se 1 rychlejší než opakované hašování a hledání volného místa, případně již uloženého záznamu, který se na kolizní adresu nepovedlo uložit. Faktor naplnění může být díky zřetězení (i násobně) větší než 1. V případě že bude seznam obsahovat významně velké množství prvků, může složitost operací hašovací tabulky degradovat až na úroveň složitosti datové struktury zvolené pro uložení množiny kolizních prvků. Vzhledem k omezenému počtu prvků, které dobře zvolená hašovací funkce přiřadí na stejnou adresu, ale nejsou výkonnostní dopady tak výrazné. Obrázek 1.3 pak ukazuje hašovací tabulku se zřetězením záznamů po vložení stejných záznamů jako v příkladu u otevřené adresace. Výhodou oproti otevřené adresaci je i snadné mazání a přehašování prvků, opět díky omezenému počtu prvků, na kterých se tyto operace provádí a také díky tomu že pracujeme pouze s prvky se stejným otiskem klíče.



Obrázek 1.3 *Zřetězení záznamů*

2 Dynamické hašování

Jedna z nevýhod hašovacích tabulek je, že potřebujeme dopředu určit velikost tabulky založenou na odhadovaném počtu vkládaných prvků. Tato nevýhoda se týká ve větší míře tabulek s otevřenou adresací, protože mají faktor naplnění vždy menší než 1, ale postiženy jsou i hašovací tabulky využívající zřetězení záznamů, protože čím větší je faktor naplnění, tím delší seznam prvků hašovaných na jednu adresu musíme projít při každé operaci. Tento problém lze vyřešit přehašováním, kdy na počátku je hašovací tabulka menší a když faktor naplnění přesáhne určitou mez, je velikost tabulky navýšena a již uložené hodnoty jsou přehašovány do nové tabulky.

Pokud budeme při přetečení zvětšovat tabulku geometrickou řadou, např. vždy na dvojnásobek, bude celková očekávaná amortizovaná složitost operací konstantní (tj. O(1)), i když započítáme čas všech přehašování. To znamená, že pokud rozpočítáme čas přehašování na jednotlivé prvky, bude příspěvek jednoho prvku konstantní, tj. čas nezávisí na počtu prvků ve struktuře (ani na počtu postupných přehašování). Podobným způsobem lze tabulku zmenšovat na polovinu, pokud počet prvků podteče 1/8 velikosti tabulky. Po přehašování má tabulka 1/4 - 1/2 prvků a do dalšího přehašování se nutně vykoná dost operací, které ho amortizovaně zaplatí.

Implementačně lze přehašování realizovat dávkově, kdy pozastavíme program, který tabulku využívá, a vše naráz přehašujeme. Druhá metoda je postupné přehašování bez zastavení programu, kdy současně nějakou dobu používáme starou i novou tabulku. Při každé operaci přehašujeme několik prvků a až jsou všechny prvky přemístěny, starou tabulku dealokujeme.

Třetí možností jak zvětšit velikost hašovací tabulky, jsou algoritmy, které umožňují za běhu programu postupně zvětšovat velikost hašovací tabulky po malých krocích. Stejně tak přehašování prvků je postupné, v návaznosti na změnu velikosti tabulky. I sama hašovací funkce pak mění dle aktuální velikosti hašovací tabulky své parametry tak, aby vkládané prvky umisťovala rovnoměrně do celého adresního prostoru tabulky a zároveň aby vždy správně adresovala i prvky vložené před změnou velikosti tabulky. Takto navržené algoritmy nazýváme *dynamické hašování* (dynamic hash tables). Vzhledem k tomu, že je najednou přemisťován pouze malý počet prvků ovlivněných jedním krokem zvětšení, není ani dopad přehašování na výkon hašovací tabulky, která je stále používána, nikterak dramatický. Dále si blíže popíšeme dvě nejznámější metody dynamického hašování, jejichž implementace a testování jsou předmětem této bakalářské práce. Jedná se konkrétně o Linear hashing a Extendible hashing.

2.1 Lineární hašování

Lineární hašování (linear hashing) je algoritmem dynamické (rostoucí) hašovací tabulky využívající k řešení kolizí zřetězení záznamů. Na 6th Conference on Very Large Databases v roce 1980 jej představil Witold Litwin ve své práci "Linear hashing: A new tool for file and table addressing" [3]. Tabulka umožňuje dle potřeby růst po malých krocích, kdy se vždy přeuspořádá (přehašuje) jen jeden uzel prvků adresovaných stejnou hodnotou hašovací funkce. Na konec tabulky se přidá jeden nový uzel a přeuspořádáním se část prvků přesune z původní pozice do nového uzlu. Rozšiřování tabulky tak probíhá postupně, s minimálním výkonnostním dopadem

na běh programu a provádí se vždy po určitém počtu vložených prvků, tak aby se faktor naplnění tabulky udržoval v nastavených mezích. Přetékající uzly (uzly, jejichž počet prvků překročil kapacitu uzlu) nejsou rozděleny a přehašovány ihned v okamžik přetečení a proto je nutné po dobu než k rozdělení uzlu dojde využít mechanismu řešení kolizí řetězením uzlů. Jednotlivé uzly jsou rozdělovány v pořadí dle umístění v tabulce (i ty které ještě nepřetekly) a postupně tak dojde i na ty přetékající.

Pro implementaci lineární hašovací tabulky jsou klíčové tyto 3 hodnoty:

N Prvotní velikost tabulky. Tato hodnota je důležitá pro adresaci v tabulce a neroste s tabulkou.

L Úroveň (level) tabulky. Celočíselná hodnota udávající kolikrát byla velikost tabulky zdvojnásobena. Na počátku má hodnotu 0.

S Step pointer – ukazatel na uzel, který bude při dalším rozšiřování tabulky přehašován. Na počátku je tato hodnota 0, tj. ukazuje na první záznam v tabulce.

Tabulka je rozdělena na 3 části:

$\mathbf{0}\,..\,\boldsymbol{S}-\mathbf{1}$ Oblast hašovací tabulky, která již byla od posledního povýšení úrovně tabulky rozšířena.

$\boldsymbol{S}\,..\,\boldsymbol{N}\times\mathbf{2}^{\boldsymbol{L}}+\boldsymbol{S}-\mathbf{1}$ Oblast hašovací tabulky, která na rozšíření teprve čeká.

$\boldsymbol{N}\times\mathbf{2}^{\boldsymbol{L}}\,..\,\boldsymbol{N}\times\mathbf{2}^{\boldsymbol{L}}+\boldsymbol{S}-\mathbf{1}$ Oblast uzlů přidaných od posledního povýšení úrovně tabulky.

Rozdělení tabulky na tři výše uvedené oblasti v různých fázích růstu tabulky je graficky znázorněno na obrázku 2.1 níže.

Obrázek 2.1 Rozdělení tabulky do tří oblastí v různých fázích růstu

Dále bude podrobněji vysvětleno fungování čtyř základních operací nad datovou strukturou, bez kterých by popis algoritmu nebyl kompletní. Jedná se o tzv. CRUD operace, tedy vložení (Create), nalezení prvku (Read), aktualizace hodnoty již existujícího prvku (Update) a vymazání (Delete).

2.1.1 Create

Vkládání prvků je prosté. Voláme hašovací funkci s klíčem ukládaného prvku a funkce nám vrátí výsledek H. Na základě této hodnoty, určíme adresu v tabulce (viz. Read). Prvek vložíme do seznamu prvků se stejným otiskem. Po určitém počtu prvků přidaných do tabulky (při dosažení hranice optimálního faktoru plnění) je nutné hašovací tabulku rozšířit. Postup je následující:

1. Pokud je $S = N \times 2^L$ pak se level tabulky L zvýší o 1 a ukazatel S se vynuluje.
2. Na konec tabulky se přidá nový uzel.
3. Seznam prvků na pozici S se z tabulky vyjme a vloží se znovu, nyní se záznamy rozdělují mezi dva uzly (původní a nově přidaný), počet prvků v jednom uzlu by tak měl být nižší.
4. Ukazatel S se zvýší o 1.

2.1.2 **Read**

Na základě klíče nám hašovací funkce vrátí hodnotu H (otisk, hash value). Adresa prvku se pak určuje na základě vztahu mezi výsledkem hašovací funkce H a ukazatelem S (step pointer):

pro $H\ mod\ (N \times 2^L) < S$ je adresa $H\ mod\ (N \times 2^{L+1})$

pro $H\ mod\ (N \times 2^L) \geq S$ je adresa $H\ mod\ (N \times 2^L)$

Jinými slovy, pokud hašovací funkce adresující klíče do adresního prostoru 0..N-1 vrátí hash value menší než je step pointer S, tedy adresu z oblasti, kde už došlo k rozšíření hašovací tabulky a k přerozdělení uložených záznamů mezi původní a nové uzly, pak se jako hash value použije výsledek hašovací funkce adresující klíče do prostoru 0..2*N-1. Hašovací funkce je zkonstruována tak, že hodnoty, které by pro tabulku o velikosti N spadaly do rozsahu 0..S-1, pro tabulku o velikosti 2*N spadají do rozsahu 0..S-1 nebo N..N+S-1. Nemůže tedy dojít k adresaci za hranici N+S a paměť alokovaná pro hašovací tabulku může být zvětšována postupně.

Pak už jen stačí projít seznam prvků se stejným otiskem, dokud nenajdeme námi hledaný prvek. Pokud v seznamu hledaný prvek nenalezneme, není daný prvek v hašovací tabulce vůbec uložen. Lineární hašování nevyužívá otevřenou adresaci a prvek tak nemůže být na jiné adrese.

2.1.3 **Update**

Pokud aktualizujeme prvek tabulky a měníme hodnoty, které nemají vliv na klíč, pak stačí prvek pouze najít a požadované hodnoty upravit. Pokud se mění hodnoty ovlivňující klíč (a tedy i výsledek hašovací funkce), je nutné znát původní hodnotu klíče a uložený prvek najít a vyjmout z tabulky. Po úpravě požadovaných položek se na základě těchto nových hodnot pak prvek uloží na nové místo do tabulky (přehašování). Protože lineární hašování nemění velikost tabulky na základě obsazenosti jednotlivých uzlů, ale podle faktoru naplnění, není zapotřebí při přehašování prvku měnit velikost tabulky.

2.1.4 **Delete**

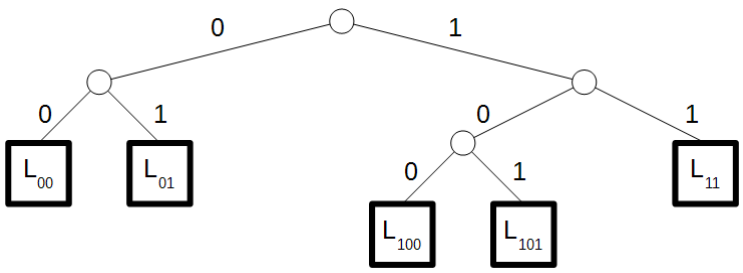
Prvek je nutno nejprve nalézt (operace Read) a pak se pouze vyjme ze seznamu prvků se stejným otiskem. Odstranění samotné závisí na použitém způsobu uložení záznamů uvnitř uzlu. Pokud odstraníme podstatné množství prvků v tabulce a faktor naplnění tabulky klesne pod určitou mez, můžeme tabulku zmenšit. Postup je analogický k rozšíření tabulky:

1. Pokud je S=0 a L=0, tabulka je na své původní velikosti a dále se nezmenšuje.
2. Pokud je S=0 a L>0, pak se L zmenší o 1 a S se nastaví na hodnotu $N \times 2^{L-1}$.
3. Pokud bylo S>0, pouze se zmenší o 1.
4. Prvky z adresy $\boldsymbol{N \times 2^{L} + S - 1}$ se přesunou na adresu S-1.
5. Odstraní se uzel na konci tabulky.

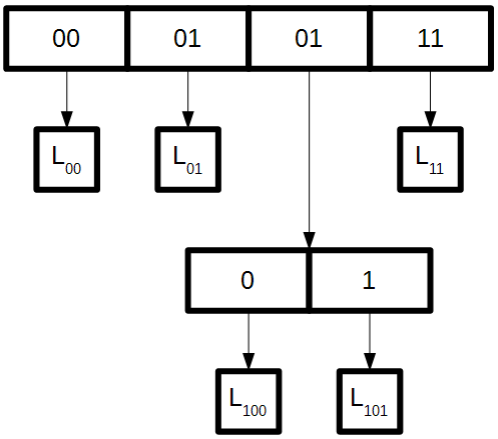
Praktičtější může být zmenšování tabulky o výraznější množství uzlů, jak je to zdůvodněno v kapitole Rostoucí tabulky a přehašování. Vyhneme se tím časově náročné realokaci paměti a kopírování dat při opakovaném zvyšování a snižování úrovně tabulky. V takovém případě je prvním krokem snížení ukazatele S na nulu a přesunutí obsahu všech uzlů přidaných od posledního povýšení úrovně tabulky L (všechny uzly za adresou $N \times 2^{L-1}$) do odpovídajících uzlů na adrese o $N \times 2^{L}$ nižší. Uvolněné uzly pak mohou být smazány. Pokud by takové zmenšení tabulky nebylo dostatečné, může být dalším krokem snížení úrovně tabulky L o 1, čímž dojde ke zmenšení tabulky na polovinu.

2.2 Extendible hašování

Odlišný přístup k dynamickým tabulkám zvolili o rok dříve Fagin, Nievergelt, Pippenger a Strong v práci nazvané Extendible Hashing – A Fast Access Method for Dynamic Files [2]. Hašovací funkce h() mapuje prostor klíčů S na prostor adres A. Jedním z velkých problémů hašování je docílit rovnoměrné distribuce mapování uložených klíčů do prostoru adres A, tak aby nedocházelo ke shlukování klíčů, které vede k přetékání uzlů s uloženými záznamy nebo k vytváření dlouhých souvislých obsazených oblastí při použití otevřené adresace. *Rozšiřitelné hašování* (extendible hashing) používá k uložení záznamů uzly o pevně dané velikosti (typicky odpovídají velikosti bloku dat – stránce paměti nebo clusteru zařízení použitého k perzistentnímu uložení dat). Přetékání řeší rozdělením adresního prostoru A na m částí. Tyto části pak podle jejich obsazenosti může dále dělit, nebo slučovat, obsah přetékajícího uzlu po rozdělení adresního prostoru původního uzlu přehašuje mezi dva nové. Pro rychlou adresaci prostoru rozděleného na nestejné části napodobuje chování *samovyvažovacích stromů* (obrázky 2.2 a 2.3), kde jednotlivé úseky adresního prostoru A jsou listy tohoto stromu. Současně se snaží obejít hlavní nevýhodu – postupné několikanásobné vyhodnocování klíče než je dosaženo hledaného listu stromu. Využívá k tomu mapování stromu hašovacích adres do tabulky, v originále označované *directory* (adresář).

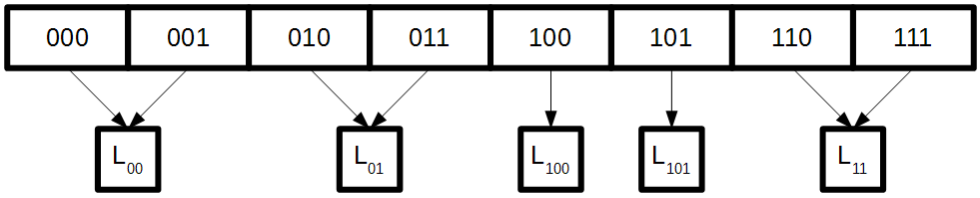


Obrázek 2.2 *Strom*



Obrázek 2.3 *Víceúrovňové mapování stromu do tabulky*

Za cenu mírně vyšší paměťové režie, kdy např. kódy s prefixem 000 a 001 odkazují na stejný list (obrázek 2.4), je dosaženo dostupnosti všech listů v jednom kroku. Adresář v příkladu k rozdělení adres využívá první tři bity výsledku hašovací funkce. To je označováno jako *globální hloubka* (global depth) adresáře D=3 (k rozlišení že záznamy začínající 100 a 101 patří do uzlů L_{100} a L_{101} slouží 3 bity adresy). Pokud je v některých uzlech málo záznamů, může dojít ke sloučení, jako ve výše uvedeném příkladu, kdy pro uložení záznamů začínajících 000 a 001 je použit pouze jeden uzel L_{00}. Stejně jako záznamy 010 a 011 jsou uloženy ve společném uzlu L_{01}. K určení že záznamy s prefixem 000 a 001 patří do L_{00} stačí pouze první dva bity adresy. Uzel L_{00} má tedy definovanou *lokální hloubku* (local depth) d=2. Globální hloubka adresáře D nemůže být menší než kterákoliv lokální hloubka d.



Obrázek 2.4 *Převedení do jedné tabulky*

Pokud bychom se do rozšiřitelné hašovací tabulky znázorněné příkladem výše pokusili uložit záznam, jehož adresa bude začínat prefixem např. 010 a zjistíme, že uzel L_{01} je již plný, rozdělíme L_{01} na dva nové uzly L_{010} a L_{011}. Na původní uzel vedly z adresáře dva odkazy z pozic 010 a 011. Nově bude každé této pozici přiřazen vlastní uzel s lokální hloubkou d=3 a záznamy původně uložené v L_{01} vyjmeme a přehašujeme mezi L_{010} a L_{011}. Tím by se nám měla snížit (v ideálním případě na polovinu) obsazenost nových uzlů a nový záznam již bude možno vložit. Toto rozdělení je však možné, jen pokud je lokální hloubka (d=2) plného uzlu menší než globální hloubka adresáře (D=3). Pokud by přetekl např. uzel L_{101}, zjistíme, že na něj ukazuje pouze jedna pozice v adresáři a nemáme možnost dále rozdělit adresní prostor, abychom mohli pokračovat

a přidat nový uzel. V takovém případě je nutné zdvojnásobit velikost adresáře a k mapování listů stromu se použije více bitů z prefixu adresy, tedy globální hloubka D se zvýší o jedno. Při tomto zdvojnásobení velikosti tabulky však není nutné přehašovat všechny záznamy. Pouze se jednoduše přepočítá adresář v souladu s novou hloubkou, kdy např. na L_{00} budou ukazovat čtyři pozice (0000 až 0011) a na námi sledovaný přetékající uzel L_{101} budou ukazovat adresářové pozice 1010 a 1011. Lokální hloubka uzlu (d=3) bude nižší než globální hloubka adresáře (D=4) a může dojít k požadovanému rozdělení a přehašování pouze přetékajícího uzlu.

Podle autorů rozšiřitelného hašování je možno zbytek adresy bez prefixu (který je v jednom uzlu u všech záznamů logicky stejný) použít k hašování záznamu na pozici v rámci daného uzlu, čímž se při hledání vyhneme nutnosti procházet postupně všechny záznamy. Vzhledem k pevně danému maximálnímu počtu uložených záznamů v uzlu (faktor naplnění nemůže být větší než 1) a vzhledem k tomu že tento maximální počet záznamů nebude nikterak velký (řádově jednotky až desítky záznamů), je možno použít i hašování s otevřenou adresací, protože v těchto omezených objemech nejsou dopady jeho nevýhod tak velké.

Díky schopnosti rozšiřitelného hašování vypořádat se s nerovnoměrnou distribucí klíčů do adresního prostoru A, se nabízí použití hašovací funkce zachovávající pořadí, čímž získáme hašovací tabulku zachovávající pořadí na úrovni uzlů (klíče v uzlu s otiskem k budou při porovnání menší než klíče v uzlu s otiskem k+1). Pokud bychom pak samotné uzly realizovali jako *seřazené seznamy* (SortedList) a zajistili bychom přesunování prvků mezi jednotlivými zřetězenými uzly, získáme hašování, které zachovává pořadí.

2.2.1 **Create**

1. Provedeme prvních 5 kroků hledání.
2. Pokud je stránka P (uzel) plná, pokračujeme na krok 6 (vkládání).
3. Pokud je na pozici získané na konci kroku 1 volné místo, uložíme zde záznam.
4. Jinak použijeme řešení kolizí k uložení záznamu na stránce P.
5. Pokud byl záznam úspěšně uložen, pak zde končíme.
6. Víme, že stránka P je plná. Vytvoříme novou stránku P*, stránky P a P* budeme nazývat sesterské.
7. Do dočasného úložiště Q přesuneme všechny záznamy z původní stránky P včetně nově vkládaného záznamu.
8. Lokální hloubku stránek P a P* nastavíme na d'+1 kde d' je původní lokální hloubka stránky P.
9. Pokud je nová lokální hloubka stránek P a P* větší než globální hloubka adresáře, provedeme následující:
    - Zvýšíme globální hloubku adresáře o 1.

        - Zdvojnásobíme velikost adresáře a jednotlivé ukazatele updatujeme s ohledem na novou globální hloubku.
        - Místo původního ukazatele na stránku P, uložíme dva nové ukazatele na stránky P a P*.

10. Vložíme záznamy z dočasného úložiště Q do hašovací tabulky (rekurzivní volání).

2.2.2 **Read**

1. Spočítáme adresu pseudoklíče v hašovacím adresním prostoru: K' = h(K).
2. Zjistíme globální hloubku adresáře D.
3. Prvních D bitů adresy K' uložíme do proměnné r typu int.
4. Na r-té pozici adresáře najdeme ukazatel na stránku P (uzel).
5. Zbývajících s bitů pseudoklíče K' použijeme k hašování hledaného záznamu v hašovací tabulce stránky P.
6. Pokud je to nutné, řešíme kolize v rámci stránky P, dokud nenajdeme hledaný záznam.

2.2.3 **Update**

1. Pokud měníme pouze hodnoty neovlivňující klíč K, zavoláme metodu Read, změníme požadované hodnoty a zde končíme.
2. Jinak záznam vyjmeme z hašovací tabulky (Delete).
3. Uložíme nové hodnoty a přepočítáme klíč.
4. Změněný záznam znovu vložíme do hašovací tabulky (Create).

2.2.4 **Delete**

1. Vyhledáme záznam s klíčem K (hledání).
2. Pokud požadovaný záznam nebyl nalezen, ukončíme algoritmus s patřičným návratovým kódem.
3. Záznam odstraníme ze seznamu, nebo zneplatníme (dle použitého řešení kolizí).
4. Pokud součet záznamů na stránce P a její sesterské stránce P* (stránka která může být sloučena se stránkou P: prvních d-1 bitů pseudoklíče K' mají shodných) klesne pod stanovenou hranici, provedeme jejich sloučení následovně:
    - Všechny záznamy se stránek P a P* přesuneme do dočasného úložiště Q.
    - Jednu ze stránek dealokujeme, např. P*. Všechny ukazatele z adresáře, které odkazovaly na P*, nastavíme, aby odkazovaly na P.
    - Snížíme lokální hloubku stránky P o 1.
    - Vložíme všechny záznamy z dočasného úložiště Q na stránku P.

5. Pokud je lokální hloubka všech stránek menší než globální hloubka adresáře, můžeme zmenšit adresář následujícím způsobem:
    - Snížíme globální hloubku adresáře o 1.
    - Snížíme velikost adresáře na polovinu a všechny ukazatele adresáře updatujeme v souladu s novou globální hloubkou.

3 Praktická implementace dynamického hašování

Hlavní část této bakalářské práce spočívá v implementaci lineárního a extendible hašování do frameworku RadegastDB (dříve QuickDB) který je vyvíjen na Katedře informatiky Fakulty elektrotechniky a informatiky Vysoké školy báňské – Technické univerzity Ostrava. Framework je objektový, je vyvíjen v programovacím jazyce C++ a využívá techniku šablon tříd (Template class), umožňující vytvořit algoritmus/datovou strukturu pro obecný typ <T>, který může být nasazen na různé konkrétní datové typy. Vytvořená hašovací tabulka pak může být využita např. pro ukládání záznamů s klíčem tvořeným n-ticí celých čísel (cTuple). Stejná šablona může být v jiném projektu využita jako hašovací tabulka s klíči typu string, double nebo třeba int.

Pojďme se podívat na některá specifika hašování zachovávajícího pořadí. Hašovací funkce klíče vrací 32-bitové číslo. Pro adresaci uvnitř tabulky o velikosti 2^k je použito právě k bitů. Protože od této implementace očekáváme zachovávání pořadí, je z celého otisku klíče použito prvních k klíčů, které jsou nejdůležitější. Zde však narážíme na další problém, který u běžného hašování neznáme. Základním předpokladem úspěšné implementace je požadavek:

Pro každé dva různé klíče K_1 a K_2 a jejich otisky h_1 a h_2 platí:

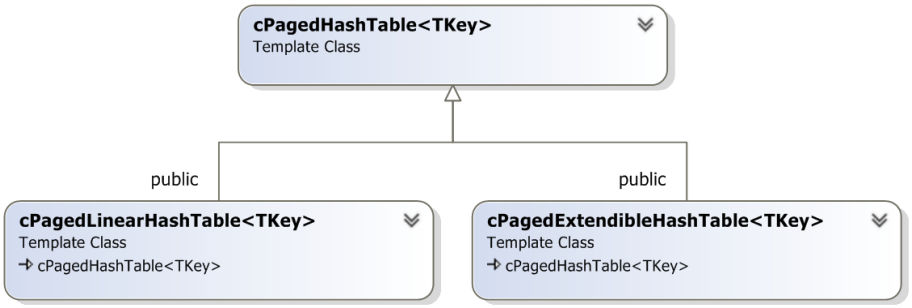
Pokud $K_1 > K_2$ pak $h_1 \geq h_2$.

Tento požadavek vylučuje použít v hašovací funkci jinak velmi oblíbenou operaci *mod* - zbytek po celočíselném dělení, která ve spolupráci s bitovou rotací pomáhá dosáhnout využití všech bitů otisku. U hašovací funkce zachovávající pořadí musí být v otisku dost místa i pro "velké" klíče. Velké myšleno nikoliv velikostí v paměti, ale v porovnání s ostatními klíči. Hašovací funkce připravena na tyto "velké" klíče, pak logicky pro "malé" klíče bude generovat otisky, jejichž nejvýznamnější bity budou mít hodnotu nula. Pokud se však rozhodneme do tabulky uložit množinu pouze malých klíčů, může se stát že z prvních k bitů otisku použitých k adresaci v rámci tabulky ponese informaci k rozlišení jednotlivých klíčů jen pár bitů s menší váhou a vyšší v tabulce budou neobsazeny. Taková hašovací tabulka nepracuje efektivně, protože naplněna jen velmi řídce a většina záznamů je uložena v nemnoha dlouhých řetězcích seřazených seznamů uzlů. Již první měření implementovaných algoritmů ukázala, že tento problém u některých datových kolekcí opravdu nastává a výrazně snižuje efektivitu datové struktury. Bylo proto dodatečně implementováno řešení, které v případě, že dopředu známe povahu vkládaných klíčů a rozsah jejich prvků, umožní využít bitový rozsah otisků násobně efektivněji. Ve třídě `cPagedHashTable` byl implementován atribut `mHashValueTrim`, který je nastaven dle velikosti tabulky a který ve funkci `GetHashValue()` zaručuje použití právě prvních *k* bitů vypočítaného otisku. Dodatečně byl doimplementován další atribut `mHashValueBonus`, který určuje kolik méně významných bitů, které by byly jinak ořezány, bude z otisku použito navíc. Výsledný otisk tak neobsahuje pouze prvních k bitů, ale `k` + `mHashValueBonus` bitů. Otisky, jejichž hodnota ukazuje díky tomuto rozšíření za adresní prostor tabulky, jsou pak všechny mapovány na poslední pozici v tabulce. Tím se může celý problém hromadění dlouhých řetězců

uzlů přenést ze začátku tabulky na její úplný konec. Správně zvolená hodnota mHashValueBonus však přináší znatelné zlepšení výkonu hašovací tabulky.

3.1 Statické hašování - cPagedHashTable

Framework již obsahoval základní implementaci hašovací tabulky, dále ji budu nazývat statickou (opak dynamicky se zvětšující), s řešením kolizí zřetězením záznamů. Implementovaná třída je pojmenovaná `cPagedHashTable` a jako uzel pro uložení záznamů využívá třídu `cNode`, konkrétně jejího potomka `cPagedHashTableNode`, který má zděděno perzistentní ukládání dat a funguje jako seřazený seznam s vyhledáváním půlením intervalů. Kombinace `cPagedHashTable` a `cPagedHashTableNode` je funkční, má zatím implementováno vkládání dat (i duplicitních položek se stejným klíčem) a jejich vyhledávání. Třída `cPagedHashTableNode` má definován *atribut* (member variable) `mNextNode`, ale `cPagedHashTable`, v době kdy jsem s vlastní implementací začínal, neuměla řetězit přetékající uzly a záznamy, které se nevešly do prvního uzlu, neuložila vůbec.



Obrázek 3.1 *Rodičovská třída hašovací tabulky a její potomci*

Třídu `cPagedHashTable` jsem doplnil o fungující řetězení uzlů a použil ji jako rodičovskou třídu pro nově vytvořené dynamické hašovací algoritmy - viz obrázek 3.1. Bylo nutné některé metody rozdělit na menší uzavřené funkční celky pro zvýšení znovupoužitelnosti kódu, aby třídy potomků nemusely znovu tvořit celé metody i když se algoritmus liší jen v pár krocích. Většinu metod rodičovské třídy `cPagedHashTable` jsem také doplnil o modifikátor virtual, který umožňuje polymorfismus. Tedy přepsat v potomkovi (dílčí) metody rodičovské třídy tak, že když instance třídy potomka zavolá metodu rodičovské třídy, budou (tyto dílčí) metody volané z metody rodiče, pokud jsou deklarovány jako virtual, hledány v třídě potomka a pokud je potomek redefinoval, bude vykonána tato aktualizovaná verze. To mi umožnilo použít softwarový návrhový vzor Template method (šablona metody). Kupříkladu hledání záznamů - metoda `PointQuery()` je tak definováno pouze v rodičovské třídě `cPagedHashTable` a veškeré rozdíly ve vyhledávání v Lineární a Extendible hašovací tabulce jsou implementovány pouze změnou jediné metody `GetHashValue()`. Pojďme se blíže seznámit s atributy a metodami této třídy klíčovými pro implementaci hašování a pro samotné testování.

Začneme důležitými atributy. Pole `mHashArray` slouží k uchovávání odkazů na jednotlivé uzly. Je deklarováno jako pole uživatelského typu `tNodeIndex`, který je reprezentován 32 bitovým typem unsigned int. V každé položce je uložen index (id) přiřazeného uzlu, pokud adresa v poli nemá přiřazen žádný uzel, je uložena hodnota `EMPTY_LINK` definovaná jako (unsigned int) -1. Třída `cPagedHashTable` velikost `mHashArray` během své existence nemění. Nastavená velikost je uložena v atributu `mSize`. Dále následují dva atributy používané pro statistiku parametrů datové struktury za běhu programu. Počet uzlů odkazovaných přímo z hašovací tabulky je uložen v `mRootNodes`. Z poměru `mRootNodes/mSize` lze určit kvalitu distribuce hašovací funkce. Hodnoty blížící se 1 značí výbornou distribuci záznamů napříč adresním prostorem hašovací tabulky.

```
template<class TKey>
class cPagedHashTable
{
protected:
typedef typename cPagedHashTableNode<TKey> tNode;

tNodeIndex *mHashArray;
tNodeIndex  mSize;      // HT base size N
tNodeIndex  mRootNodes; // nodes attached directly to the hash table
static const uchar mHashValueBits = 8 * sizeof(tNodeIndex);
// Bit size of value used for hash (32 for uint)

uchar       mHashValueTrim;
uchar       mHashValueBonus;

virtual inline tNodeIndex GetNodeIndex(const TKey &key);
virtual inline tNodeIndex GetNextNodeIndex(const tNode* node) const;
virtual inline tNodeIndex GetHashValue(const TKey &key);

tNode* ReadNewNode();
tNode* ReadNodeW(unsigned int index);
tNode* ReadNodeR(unsigned int index);
inline void GetNodeW(tNodeIndex &nodeIndex, tNode* &node);
inline int InsertIntoSortedNodeChain(tNodeIndex nodeIndex,
      unsigned int &chaining, const TKey &key, char* data,
      bool pInsertOrUpdate = false);

public:
virtual int Insert(const TKey &key, char* data,
      bool insertOrUpdate = false);
virtual cTreeItemStream<TKey>* Find(const TKey &key, char* data);
virtual bool PointQuery(const TKey &key, char* pData);
```

```
virtual inline unsigned int SetHashTableSize(tNodeIndex newSize);
virtual inline unsigned int GetRealHashTableSize();
inline unsigned int GetHashTableSize();
inline unsigned int GetRootNodes() const;
inline uchar SetHashValueBonus(uchar newBonus);

unsigned int mHashBonusFails;
// counter of HashValue overruns due to mHashValueBonus
unsigned int mMaxNodeChainLength;
// length of the longest chain found during Insert()

void DoHashTableStatistics(uint &TableSize, uint &TotalNodes,
      uint &RootNodes, uint &TotalItems, uint &LongestChain);
};
```

Nízký počet kořenových uzlů naopak svědčí o špatném rozprostření a v tabulce může docházet k výskytu dlouhých řetězců uzlů se stejným otiskem. Taková hašovací tabulka ztrácí své přednosti a chová se spíše jako sorted list. V pozdější části bakalářské práce, při měření a vyhodnocování výsledků, tento poměr nazývám kořenový faktor naplnění, případně anglicky - root fill (ratio). Další atribut `mMaxNodeChainLength` nám pak sděluje délku nejdelšího řetězce, na který jsme narazili během vkládání. Hodnota je pouze zvyšována, odstranění záznamu ji nesníží. Podrobnější údaje o parametrech hašovací tabulky můžeme získat pomocí metody `DoHashTableStatistics()`, která projde celou tabulku a všechny uzly a vrátí zjištěné počty uzlů, záznamů a aktuální délku nejdelšího řetězce uzlů. Ta může být nižší než `mMaxNodeChainLength`. Projít celou hašovací tabulku a všechny uzly je časově náročné a není určeno k provádění během normálního provozu datové struktury.

Dále jsou tady podpůrné metody a atributy pro hašování a práci s uzly. Atribut `mHashValueBonus` jsme si popsali již v úvodu třetí kapitoly. Jeho hodnota je nastavována metodou `SetHashValueBonus()`. Tato metoda upravuje také hodnotu atributu `mHashValueTrim`, který je jinak nastavován metodou `SetHashTableSize()`. Atribut určuje, o kolik bitů má být bitově posunut doprava vypočítaný otisk klíče, aby výsledek odpovídal velikosti tabulky. Metoda `GetHashValue()` vrací výsledný otisk klíče. Metoda `GetNodeIndex()` je jejím rozšířením a vrací k danému klíči přímo index příslušného uzlu. Pokud je uzel plný, vrátí nám index dalšího (zřetězeného) uzlu metoda `GetNextNodeIndex()`. Za zmínku stojí i `ReadNewNode()` která nám vrací nový, prázdný uzel. Metoda využívá třídu `cPagedHashTableHeader` která udržuje seznam uzlů, které byly již vytvořeny, ale aktuálně se nepoužívají a jsou prázdné. V původní implementaci se s vracením uzlů nepočítalo a vždy se vygeneroval nový. Tuto vlastnost využívají především potomci - třídy dynamického hašování při dělení uzlů, ale k tomu se dostaneme později.

Zbývají nám ty nejdůležitější metody - `Insert()` pro vkládání a `PointQuery()`, `Find()` pro vyhledávání uložených prvků. Vkládání probíhá následovně. Metoda `Insert()` zjistí otisk daného klíče - adresu v hašovací tabulce. Pokud uzel zatím nebyl vytvořen, vytvoří nový, odkaz na něj vloží do tabulky, uloží do něj prvek a zde funkce končí. Jinak zavolá funkci

`InsertIntoSortedNodeChain()` která se postará o vložení do uzlu a pokud je uzel zřetězený (není jediný) zařídí tato metoda průchod řetězcem uzlů, nalezení správného uzlu ke vložení, vložení nového prvku a obhospodaří i přetékání prvků mezi uzly které nastane při vkládání nového prvku do plného uzlu. Toto vkládání je úmyslně vyčleněno do separátní funkce. Třídy potomků redefinují metodu `Insert()` a hledání toho správného uzlu, případně dynamické zvětšení tabulky, ale když dojde na samotné vkládání do řetězu uzlů, je postup stejný a definovaný již v rodičovské třídě. Podobné je to s metodou `PointQuery()` která vyhledá a vrátí data již uložená v hašovací tabulce. Zde je postup pro všechny tři varianty hašovací tabulky natolik podobný, že třídám potomků stačí redefinovat metodu `GetHashValue()` a s vráceným otiskem pak rodičovská třída najde správný kořenový uzel a projde zřetězené uzly, dokud nenalezne a nevrátí požadovaná data, nebo chybový kód že hledaný klíč nebyl nalezen.

3.2 Lineární hašování - cPagedLinearHashTable

Dostáváme se k implementaci první dynamicky rostoucí hašovací tabulky. Dle Litwina nemusí být oblast pro nové/rozdělené uzly součástí bloku paměti alokovaného pro samotnou hašovací tabulku (zde `mHashArray`). Já jsem však zvolil variantu, kdy rovnou alokuji `2*mSize` a oblast rozšíření plynule navazuje na hašovací tabulku. Zde nabývá na důležitosti metoda `GetRealHashTableSize()`, která na rozdíl od rodičovské třídy vrací rozdílné výsledky než `GetHashTableSize()`. Druhá jmenovaná vrací velikost základní hašovací tabulky a tento údaj použijeme například při alokaci paměti pro větší tabulku. Hodnotu `GetRealHashTableSize()`, tedy velikost tabulky včetně již použité oblasti pro rozšíření naopak použijeme třeba při výpočtu faktoru naplnění. Dalším rozdílem oproti Litwinovi je v úvodu kapitoly popsaná nemožnost použít pro generování otisku správné délky operaci zbytku po celočíselném dělení. Výsledkem Litwinova algoritmu, v okamžiku kdy step pointer dosáhl konce základní tabulky, je už přímo nová tabulka dvojnásobné velikosti, poněvadž prvky v druhé části tabulky jsou adresovány otiskem delším o 1 bit. Tento bit navíc je u Litwinovy varianty nejvíce významným bitem a záznamy na konci tabulky, jejichž otisky mají tento bit nastaveny, jsou již na správné pozici v adresním prostoru nové tabulky. Oproti tomu implementovaná varianta přidává navíc bit nejméně významný. V okamžiku, kdy step pointer dorazí na konec základní části tabulky, jsou tak záznamy rozděleny na ty se sudým otiskem v první polovině tabulky a ty s lichým otiskem v druhé polovině. Při povýšení tabulky na další úroveň je tak nutno tyto záznamy překopírovat do nové tabulky ve správném pořadí.

```
template<class TKey>
class cPagedLinearHashTable : public cPagedHashTable<TKey>
{
protected:
tNodeIndex mSplit; // S pointer - points to the next bucket to split
bool       mForceSplit; // forces split when set to true

inline int SplitNode();
inline int RaiseLevel();      // Raises level of HT by doubling it
virtual inline tNodeIndex GetHashValue(const TKey &key);
```

```
public:
virtual int Insert(const TKey &key, char* data,
      bool insertOrUpdate = false);
virtual inline unsigned int GetRealHashTableSize();
virtual inline unsigned int GetRealHashTableSize() const;
// returns real count of all root entries available for hashing
};
```

Třída `cPagedLinearHashTable` definuje dva nové atributy. První je `mSplit`, který plní funkci step pointeru S známého z teoretického popisu Lineárního hašování. Druhým přírůstkem je `mForceSplit` typu bool, který, jak název napovídá, slouží jako indikátor, že má dojít k dílčímu zvětšení tabulky. Tento indikátor je nastaven např. během vkládání nového prvku, pokud je zjištěno příliš dlouhé zřetězení uzlů. Rozdělení uzlu, realizované metodou `SplitNode()`, je pak provedeno před vložením dalšího prvku. Pokud `SplitNode()` zjistí, že ukazatel `mSplit` již ukazuje na konec oblasti pro rozšíření, je zavolána metoda `RaiseLevel()` která zdvojnásobí velikost tabulky a provede překopírování dat. Pokud již není kam růst, `RaiseLevel()` vrátí chybový kód a algoritmus se nadále spoléhá pouze na řetězení uzlů. Limitem růstu tabulky je počet bitů otisku klíče. Dalším omezením pak může být využití atributu `mHashValueBonus`, jenž má za následek de facto ignorování prvních několik bitů otisku. Na závěr zmíním metodu `Insert()`, která se od rodičovské varianty liší pouze přidáním vyhodnocení potřeby a případně zpracováním požadavku na rozdělení dalšího uzlu. Část kódu vyhledávající vhodný uzel, stejně jako následné volání `InsertIntoSortedNodeChain()` zůstává prakticky shodná.

3.3 Rozšiřitelné hašování - cPagedExtendibleHashTable

Extendible hašování přináší ve srovnání s předchozím lineárním hašováním větší změny oproti své rodičovské třídě. Přidáno je pole mLocal udržující seznam lokálních hloubek jednotlivých kořenových uzlů hašovací tabulky a taky atribut `mGlobal`, ve kterém je uložena globální hloubka celé hašovací tabulky. Pro práci s hloubkami adresace musely být upraveny i metody `GetHashValue()` a `GetNodeW()` a přibyla nová metoda `AddNewNode()` která zajišťuje správné přidávání uzlů do hašovací tabulky, tak aby na nově přidaný uzel odkazovaly všechny adresy pro které je společný. Na rozdíl od lineárního hašování, které rozděluje uzly postupně, dle pořadí v hašovací tabulce a s řetězením přetékajících uzlů, na které ještě nepřišla řada, se počítá od začátku, extendible hašování řeší přetékající uzly okamžitě a řetězení je až náhradní možnost když už hašovací tabulku nelze dále zvětšit. V této implementaci je hranice rozšiřování tabulky omezena počtem bitů otisku, což dává hašovací tabulku až o 2 miliardách záznamů typu unsigned int, tedy 8GB paměti pouze pro uložení hašovací tabulky samotné (pokud není využit `mHashValueBonus`, který stejně jako u lineárního hašování snižuje délku otisku využitelnou pro adresaci). Toto omezení je společné všem třem implementacím hašování, extendible hašování ale díky tomu, že ve své čisté formě s řetězením uzlů vůbec nepočítá, má na stejných datech k dosažení limitu 32 bitové adresace ze všech tří implementovaných hašování ty největší předpoklady.

```
template<class TKey>
class cPagedExtendibleHashTable : public cPagedHashTable<TKey>
{
protected:
uchar *mLocal;    // Local level of Extendible HT
uchar mGlobal;    // Global level of Extendible HT
uchar mMaxGlobal; // no more than 32 bits for hash table index

inline void AddNewNode(tNodeIndex hash, tNodeIndex node);
inline int SplitNode(tNodeIndex full_hash, tNode* &node);
inline int RaiseLevel(); // Raises global level of HT by doubling it

virtual inline tNodeIndex TrimHashValue(tNodeIndex hash);
virtual inline tNodeIndex GetHashValue(const TKey &key);
virtual inline tNodeIndex GetHashValue(const TKey &key,
      bool trimHash);

public:
virtual int Insert(const TKey &key, char* data,
      bool insertOrUpdate = false);

virtual inline unsigned int SetHashTableSize(unsigned int size);
inline unsigned int SetMaxGlobal(unsigned int maxGlobal);
inline unsigned int GetMaxGlobal() const;

virtual inline void GetNodeW(tNodeIndex hash, tNodeIndex &nodeIndex,
      tNode* &node, bool chaining = false);
};
```

Také metoda `Insert()` byla u extendible hašování zásadně přepracována a ne jen drobně doplněna jako tomu bylo u lineárního hašování. Zde už nestačí pouze vypočítat adresu a vložit nový prvek. Objevuje se tady cyklus, který opakovaně vyhledává cílový uzel, dokud v něm není volné místo a volání metody `SplitNode()` pro rozdělení uzlu je zde přímou součástí tohoto cyklu. Jak už bylo popsáno výše, zřetězení uzlů je u této varianty hašování až únikovým východiskem v případě nadměrného výskytu záznamů se stejným otiskem. Tato konkrétní implementace pak obsahuje ochranný mechanismus, který při velmi nízkém kořenovém faktoru naplnění zabraňuje dalšímu růstu tabulky do extrémních rozměrů, jenž by vzhledem ke koncentraci otisků klíčů do úzkého úseku adresního prostoru problém stejně nevyřešil. Tento ochranný mechanismu byl do algoritmu přidán dodatečně po prvních testech, společně s `mHashValueBonus` atributem.

4 Měření rychlosti a vyhodnocení výsledků

4.1 Metodika měření

Největší výhodou hašovacích tabulek je rychlost přístupu k datům, v ideálním případě najde hašovací algoritmus hledaný klíč na první pokus, složitost je tedy O(1) a tuto vlastnost by měla hašovací tabulka správně zvolené velikosti vykazovat až do hodnot faktoru naplnění blížících se 1. Změříme tři naimplementované hašovací tabulky a ověříme, nakolik jsme se tomuto cíli přiblížili.

Na dosažení dobrých výsledků má u hašovacích tabulek klíčový význam dobře fungující hašovací funkce. Na rozdíl od běžných hašování, u hašování zachovávajícího pořadí je už z podstaty algoritmu jasné, že hašovací funkce si neporadí tak dobře s distribucí podobných klíčů, které nemůže rozprostřít napříč celou tabulkou. Jak jsem již dříve zmínil v části věnované implementaci, první testy odhalily, že vracené otisky klíčů u některých datových kolekcí začínají skupinou nulových bitů, které znemožňují rozprostření záznamů po celé ploše tabulky, zatímco méně významné bity otisku jsou ořezávány tak, aby výsledná hodnota nepřesáhla adresní prostor tabulky.

První test tak bude zaměřen na porovnání vlivu atributu `mHashValueBonus`. Bude změřena rychlost vkládání a hledání záznamů hašovacích tabulek s tímto atributem nulovým a posléze nastaveným na hodnotu, která dává nejlepší výsledky. V dalších měřeních už budu pracovat pouze s `mHashValueBonus` nastaveným na optimální hodnotu.

Druhá skupina měření bude porovnávat implementované algoritmy mezi sebou. Provedu měření statické hašovací tabulky velikosti 64k (2^{16} položek) a 1M (2^{20} položek), spolu s dynamickým hašováním lineárním i rozšiřitelným (obojí bude začínat na velikosti 64k). Pro srovnání budou použity hodnoty naměřené na stejných datech u R-stromu a B-stromu (binary tree).

Třetí oblastí, na kterou se zaměřím, je škálovatelnost. Bude nás zajímat, jak se mění výkon hašovacích tabulek v závislosti na počtu vložených prvků a na distribuci otisků klíčů. Vyberu dvě datové sady s co největším počtem záznamů, tak aby jedna měla záznamy distribuovány co nejlépe a druhá co nejvíce na jednom místě, a ověříme, zda se složitost algoritmu, u dobře rozložených záznamů, blíží O(1), a zda u lokálně situovaných záznamů opravdu stoupá složitost k hodnotám, které odpovídají druhotné datové struktuře – lineárnímu seznamu uzlů.

Čtvrté měření bude zkoumat vliv dynamického zvětšování tabulky na její výkon. Vyberu vhodnou kolekci dat a otestuji ji na dynamických hašovacích tabulkách s malou počáteční velikostí a následně s počáteční velikostí rovnající se konečné velikosti tabulky z prvního měření. Rozdíl v naměřených výsledcích bude způsoben právě opakovaným postupným přehašováním záznamů.

Všechna měření jsou prováděna s vyrovnávací paměti uzlů nastavenou na velikost 1 125 000 uzlů, což je dostatečné množství, které pojme všechny uzly. Součástí měření tak nebudou přístupy na disk, které by do měření zanesly chybu řádově přesahující rozdíly mezi jednotlivými algoritmy a porovnání jednotlivých algoritmů by znehodnotily. Datová velikost uzlu byla nastavena u všech metod shodně na 2048 bajtů. V popisu tabulek a grafů s výsledky jsou jednotlivé metody značeny dle tabulky 2.

Tabulka 2 *Značení datových struktur ve výsledcích testů*

EXT	Rozšiřitelné (extendible) hašování
LIN	Lineární hašování
STD	Statická (nerostoucí) hašovací tabulka se zachováním pořadí. Bude uváděno společně s velikostí tabulky - např. 64k STD pro tabulku o velikosti 65536 položek.
R-Tree	R-strom
B-Tree	Binární strom (B-strom)

4.1.1 Konfigurace testovací sestavy

Veškeré testování probíhalo univerzitním serveru dbedu.cs.vsb.cz s níže uvedenou HW konfigurací a softwarovým vybavením.

Tabulka 3 *Konfigurace testovací sestavy*

Procesor	2x Intel® Xeon® X5670 s pracovní frekvencí 2.93 GHz *Pozn.: Testované algoritmy* nemají *implementován*u *podporu vláken a nedokáží využít více než jedno procesorové jádro.*
Operační paměť	96 GB RAM *Pozn.: Testovací aplikace byla kompilována jako 64 bitová kvůli využití velké části dostupné operační paměti.*
Pevný disk	*Pozn.: Načítání testovacích dat ani ukládání/načítání datové struktury na perzistentní úložiště není součástí měřeným operací.*
Grafická karta	*Pozn.: Testovací program je konzolová aplikace s minimálním výstupem na obrazovku.*
Operační systém	Windows Server R2 Datacenter, Service Pack 1

4.1.2 Testovací aplikace

Pro účely testování implementovaných algoritmů byla vytvořena testovací konzolová aplikace. Aplikace je rovněž napsána v C++. Testování spočívá v načtení datové kolekce do paměti a následně uložení všech jejich prvků do testované datové struktury. Zaznamenán je počet uložených a přečtených záznamů a dále čas potřebný k jejich uložení a přečtení. Z těchto údajů lze spočítat rychlost v záznamech za sekundu, kterou hašovací tabulka zvládne ukládat a číst záznamy. K dalším parametrům, které budeme sledovat, patří:

- Velikost hašovací tabulky
    - Počáteční
    - Koncová
    - Reálná – u lineárního hašování koncová velikost + step pointer
- Počet uzlů potřebných k uložení záznamů
    - Celkový – pro výpočet faktoru naplnění
    - Počet kořenových uzlů – pro kořenový faktor naplnění
- Nejdelší řetězec uzlů se stejným otiskem klíče – odhalí lokálně koncentrované klíče
- Počet bonusových bitů použitých pro zvýšení kvality otisků
- Počet záznamů postižených použitím těchto bonusových bitů

Při spouštění testovací aplikace z příkazové řádky, můžeme nastavit různorodé parametry testů, jmenovitě:

- Hašovací algoritmus - statický, lineární, rozšiřitelný – volby S, L nebo E
- Počáteční velikost hašovací tabulky – mocniny 2 a jejich násobky – kilo (1024) a mega (1024*1024), např. 8192, 512k, nebo 2M záznamů
- Datovou kolekci použitou pro test - Meteo, Poker… volby 0 až 7
- Omezení počtu záznamů, které budou načteny z datová kolekce – 50000, 2M…
- Počet bitů pro vylepšení otisku klíče

Mírně upravená verze této aplikace pak byla použita pro testování R-tree a B-tree.

4.1.3 Testovací data

Testování probíhalo na záznamech typu klíč:hodnota, kde hodnotu reprezentoval typ unsigned int a klíč byl tvořen třídou cTuple (uspořádané n-tice typu unsigned int) různého řádu. Typ unsigned int je 32 bitový. K měření byly použity kolekce dat uvedené v tabulce níže. Z kolekcí byly před testováním odstraněny duplicitní záznamy, které vnášely do měření další proměnnou a znesnadňovaly vyhodnocení měření.

Tabulka 4 *Parametry testovacích kolekcí dat*

Název	Dimenze typu cTuple	Velikost jednoho záznamu [B]	Počet záznamů v jednom cNode	Počet unikátních záznamů	Velikost všech záznamů [MB]
IP_TO_ZIP	9	40	50	1 441 818	55
Kddcup	42	172	11	909 105	149
Meteo	5	24	84	57 796 503	1 323
Poker	11	48	42	997 872	46
Tiger	2	12	167	5 885 103	67
USA ROADS	3	16	127	57 733 497	881
Xmark	3	16	127	20 532 805	313

4.2 Výsledky testů

4.2.1 Optimalizace otisku klíčů – atribut mHashValueBonus

Na základě testovací aplikace jsem vytvořil program, který načte všechny záznamy zvolené datové kolekce a spočítá jejich hash value. Aplikace zaznamenává četnost výskytů jednotlivých otisků klíčů. Pro zjednodušení jsem obor hodnot hašovací funkce rozdělil do 256 stejně velkých oblastí a zaznamenával jsem četnost výskytů otisků v každé z těchto oblastí. Pokud to bylo specifikováno v parametrech při spuštění, aplikuje se na otisk i `mHashValueBonus`. V tabulce 5 jsou pro jednotlivé datové kolekce vypsány zjištěné optimální hodnoty atributu `mHashValueBonus` a prcentuální zlepšení kterého použitím této optimalizace bylo dosaženo na rozšiřitelné hašovací tabulce. Histogram hash value před a po optimalizaci je k vidění jako příloha A1.

Tabulka 5 *Zlepšení dosažené optimalizací hash value*

	Datová kolekce	Hash value bonus	Kořenové uzly	Nejdelší řetěz uzlů	Rychlost operace Create	Rychlost operace Read
Před úpravou	IP_TO_ZIP	0	51 504	4	2 626 262	5 340 067
	Kddcup	0	9 666	5 662	20 189	30 824
	Meteo	0	7 606	1 364	8 406	11 033
	Poker	0	643	14	401 606	813 008
	Tiger	0	50 126	1	1 290 876	2 150 988
	USA ROADS	0	516 964	3	1 634 260	2 545 756
	Xmark	0	280 722	1	1 599 876	2 809 121

Optimalizováno	IP_TO_ZIP	0	51 504	4	2 626 262	5 340 067
	Kddcup	9	29 239	5 498	27 179	41 233
	Meteo	1	10 156	935	11 656	14 528
	Poker	14	10 296	1	1 351 351	1 754 386
	Tiger	5	50 415	2	1 448 819	2 160 464
	USA ROADS	7	700 487	1	2 458 000	4 646 933
	Xmark	6	279 403	1	1 734 043	2 955 636
Zlepšení v %	IP_TO_ZIP	-	100	100	100	100
	Kddcup	-	302	97	135	134
	Meteo	-	134	69	139	132
	Poker	-	1 601	7	336	216
	Tiger	-	101	200	112	100
	USA ROADS	-	136	33	150	183
	Xmark	-	100	100	108	105

4.2.2 Porovnání jednotlivých datových struktur

Tohle je nejdůležitější měření ze všech testů v této bakalářské práci. Srovnává rychlost vkládání a vyhledávání záznamů dosahovanou různými datovými strukturami. Testovány byly rozšiřitelné hašování, lineární hašování, statické hašování s tabulkami o velikosti 2^{16} a také větší 2^{19}. U datových kolekcí Kddcup, Meteo a USA ROADS pak větší hašovací tabulka měla pro dosažení lepších výsledků 2^{20} položek. Pro srovnání byly testovány také stromy, konkrétně *binární strom* (B-tree) a R-strom. Tyto algoritmy již ve frameworku RadegastDB implementovány jsou a pro účely testování byla pouze upravena testovací aplikace tak, aby místo hašovací tabulky použila k ukládání záznamů strom. Stromy byly použity ve své výchozí konfiguraci, parametry datové struktury nebyly nijak optimalizovány. Výsledky jsou v příloze A2 prezentovány formou tabulky, skládaného sloupcového grafu (celkový výkon datové struktury: Create + Read) a skupinového sloupcového grafu kde lze snadno porovnat buď jen vkládání prvků, nebo jejich vyhledávání. Všechny výsledky jsou umístěny vždy v jednom grafu, tak aby bylo možno porovnat nejen různé algoritmy mezi sebou, ale i výsledky jednoho algoritmu v závislosti na zpracovávaných datech. Na první pohled je patrné, že hašování nabízí jedinečnou rychlost práce s daty. Pokud se zaměříme na porovnání jednotlivých metod hašování, nejrychlejší celkový výkon poskytuje statické hašování s dostatečně velkou tabulkou. Je patrné, že dynamické přehašovávání za běhu programu, ač prováděno po malých krocích, má měřitelný dopad na rychlost. Podrobnější analýzu předkládám v měření č. 4, které je na tento problém zaměřené. Je taky patrné, že rozšiřitelné hašování, které přehašovává pouze přetékající uzly je v testech rychlejší než lineární hašování, které postupně přehašuje všechny obsažené uzly. Nejpomalejších výsledků pak dosahuje statické hašování s nevhodně zvolenou tabulkou o velikosti pouze 64 tisíc položek. U datových souborů Kddcup a Meteo jsou hodnoty naměřené pro hašovací tabulky (ale i pro stromy) příliš nízké a rozdíly jsou nepostřehnutelné. To jasně demonstruje dramatický pokles výkonu hašovacích tabulek při velmi nepříznivé kombinaci klíčů vkládaných záznamů. Pohledem do tabulky zjistíme, že v nejhorším naměřeném případě klesla rychlost vkládání až na pouhých 1027 záznamů za sekundu. Zde se naplno projevuje hlavní nevýhoda hašování se zachováním pořadí – citlivost na vstupní data. Na jednu stranu, ve většině testů hašovací tabulky

několikanásobně překonávají stromy jak v rychlosti vkládání, tak vyhledávání záznamů. Na druhou stranu může dojít k degradaci výkonu hašování až do stavu, který je v praxi nepoužitelný. Je třeba si uvědomit, že nejnižší naměřený údaj 1027 vložených záznamů za sekundu se týká stavu, kdy jsou všechny uzly v cache paměti frameworku. Pokud bychom při reálném nasazení narazili na uzly, které je potřeba číst z disku, klesla by rychlost vkládání ještě výrazně níže, vždyť nejdelší zjištěný řetězec uzlů měl délku 6565 uzlů.

4.2.3 **Škálovatelnost**

Velkou předností hašovacích tabulek je konstantní přístupová doba k uloženým prvkům, nezávislá na množství vložených prvků – tedy složitost O(1). K dosažení těchto výsledků musí být použita hašovací funkce s dobrou distribucí otisků klíčů a také hašovací tabulka dostatečné velikosti, abychom předešli tvorbě dlouhých řetězců uzlů adresovaných na jedno místo v tabulce, což nám může degradovat jindy rychlou hašovací tabulku až na úroveň lineárního seznamu se složitostí O(n). Pro ověření těchto parametrů jsem provedl řadu měření na dvou největších datových kolekcích. Měřil jsem všechny datové struktury a postupně jsem zvyšoval počet zpracovávaných záznamů od 1 milionu až po obsah celé kolekce dat, což je téměř 58 milionů záznamů. Datová kolekce USA ROADS má ze všech testovaných souborů dat nejlepší distribucí klíčů, to je patrné z prvního měření (Příloha A1). Tady by měla být přístupová doba k prvkům konstantní a hašovací algoritmy by měly být rychlejší než stromy. Naopak u kolekce Meteo dochází k lokálním koncentracím otisků klíčů a tvoří se dlouhé posloupnosti uzlů, které práci značně zpomalují. Zde se očekává, že naměřený průběh vývoje přístupových časů bude nejen růst s počtem záznamů, ale výsledky hašovacích tabulek by měly být i výrazně horší než výsledky stromů.

Tabulky s časy uložení 1 milionu záznamů v závislosti na celkovém počtu záznamů pro jednotlivé datové struktury jsou k vidění na konci této bakalářské práce jako příloha A3. Výsledky potvrdily předpokládaný průběh vývoje přístupové doby. Za dobrých podmínek je čas vložení nebo nalezení 1 milionu záznamů u hašovacích tabulek konstantní. U stromů můžeme pozorovat předpokládanou složitost O(log n) i když výsledky binárního stromu se dost přibližují výsledkům, hašovacích tabulek. Pohled do tabulky nám ale odhalí že složitost s přibývajícími záznamy stále roste. U čtení záznamů z R-stromu je výsledkem poněkud neočekávaný průběh grafu, o to překvapivější že průběh nárůstu časů u vkládání prvků odpovídá očekávané logaritmické složitosti. Výsledek je průměrem z několika měření a v naměřených hodnotách nebyly pozorovány žádné extrémní rozdíly, takže tento průběh odpovídá skutečnosti. Dalším předpokladem, který jsem se snažil ověřit byla degradace hašovací tabulky na úroveň lineárního seznamu se složitostí O(n). K mému překvapení odpovídají naměřené průběhy škálování výkonu lineární hašovací tabulky na datech s nepříznivou distribucí otisků klíčů spíše ještě horšímu (kvadratickému) nárůstu složitosti. Při bližší analýze výsledků si ale povšimneme, že složitost statického hašování s malou hašovací tabulkou (64k STD), které degraduje na lineární seznam jako první, díky čemuž podává nejhorší výsledky, roste s narůstajícím počtem záznamů podle očekávání lineárně. Stejně tak statické hašování s velkou tabulkou a extendible hašování. Zdánlivě kvadratický nárůst složitosti u lineárního hašování tak hodnotím jako lineární složitost

s opožděným nástupem, s tím že, strmost přímky nárůstu přístupových časů je zpočátku zmírněna přerozdělováním přetékajících uzlů. Zde je vhodné podotknout, že u tabulky s nízkým kořenovým faktorem naplnění je nárůst velikosti tabulky zpomalován, protože benefity získané rozdělením dat koncentrovaných do pár lineárních seznamů neopodstatní jinak prázdnou tabulku o velikosti v řádu gigabajtů. Právě zpomalení růstu tabulky a s tím spojené zvýšení strmosti přímky se v grafu projevilo oním ohybem, který může budit zdání kvadratického nárůstu. Je třeba si však uvědomit, že měření probíhalo pouze v 8 bodech a křivka spojující tyto body je pouze klouzavá spojnice trendu. Při pohledu na samotné naměřené hodnoty zanesené do grafu si můžeme povšimnout, že body opravdu leží na přímce, která se láme přibližně kolem hodnoty 30 milionů vložených záznamů.

4.2.4 Vliv dynamického zvětšování tabulky na její výkon

Poslední měření si klade za úkol zjistit jak velký dopad na výkon má postupné zvětšování a přehašovávání dynamických tabulek. Měření jsem prováděl na datové kolekci USA ROADS, která má téměř 58 milionu záznamů s dobrou distribucí otisků klíčů. Záměrně jsem vybral tuto kolekci, protože u ní bude docházet k velkému přerozdělování záznamů mezi uzly a dopad zvětšování tabulky bude jasně viditelný. Datová kolekce Meteo by naopak byla pro toto měření nevhodná, protože většinu času operací se záznamy tvoří probublávání zřetězených uzlů, které je časově náročnější než přehašování a výsledky měření by vypovídaly o zcela jiných vlivech než o dynamickém zvětšování hašovací tabulky. Výsledek měření zobrazuje tabulka 5.

Tabulka 6 *Vliv dynamického zvětšování tabulky na její výkon*

	Počáteční velikost [-]	**Konečná velikost** [-]	**Čas Create** [ms]	**Čas Read** [ms]	**Dopad Create** [%]	**Dopad Read** [%]
Rozšiřitelné hašování	8 192	2 097 152	23606	12363	**116,0**	100,8
	2 097 152	2 097 152	20354	12267		
Lineární hašování	8 192	262 144	27304	14855	**132,8**	100,0
	262 144	262 144	20568	14858		

Měření ukázalo, že přehašovávání tabulek za běhu programu má nezanedbatelný vliv na rychlost ukládání nových záznamů. U rozšiřitelného hašování, kde rostou pouze přetékající uzly, narostl čas potřebný k uložení všech záznamů v průměru o 3253 ms, což je zhoršení o 16 %. Lineární hašování, které přerozděluje postupně všechny uzly, potřebovalo na růst tabulky o 6737 ms více, bylo tedy téměř o třetinu pomalejší než stejný algoritmus se správně nastavenou počáteční velikostí tabulky. Pokud očekáváme vložení velkého množství záznamů, je vhodnější tabulku už od začátku vytvořit větší. Během hledání uložených záznamů k rozšiřování tabulky nedochází, rozdíly naměřených časů jsou tedy na úrovni chyby měření.

Závěr

Naměřené výsledky potvrdily, že hašovací tabulky jsou velmi rychlé datové struktury. V porovnání s binarním stromem mohou být až 5,7× rychlejší, v porovnání s R-stromem dokonce až 36,9× rychlejší (oproti statickému hašování s tabulkou o velikosti 1 milion položek, naměřeno na datové kolekci USA ROADS). Měření škálovatelnosti prokázalo, že za příznivých podmínek je přístupová doba k datům konstantní – složitost O(1). Na druhou stranu se ve výsledcích projevila velká závislost na různorodosti klíčů a distibuci otisků klíčů do adresního prostoru tabulky a z ní vyplývající nutnost znát povahu klíčů a přizpůsobit jim hašovací funkci. V opačném případě jsou hašovací tabulky degradovány na nepříliš početnou množinu lineárních seznamů a výkon datové struktury se propadá v závislosti na počtu vkládaných prvků lineárně. Narozdíl od hašovacích tabulek, stromy podávaly sice nižší, ale o to vyrovnanější výkony. Jsou proto vhodnější pro univerzální použití.

Po zanalyzování naměřených výsledků je možné určit, kterým směrem by se měl ubírat případný další vývoj navržených algoritmů. Alfou a omegou výkonu hašovací tabulky je samotná hašovací funkce. Určitě by stálo za zvážení a otestování, zda-li by nebylo vhodnější použít hašovací funkci, která by například vracela 64 bitový otisk klíče. Poskytovala by více bitů nesoucích informaci a tím i větší prostor pro použití `mHashValueBonus` atributu spolu s velkými hašovacími tabulkami. Pokud se nám nepodaří zcela eliminovat příčinu poklesu výkonu, tedy omezení daná hašovací funkcí která musí zachovávat pořadí, nezbývá než zaměřit se na důsledky a to je vytváření dlouhých řetězců přetékajících uzlů. Jejich počet by se dal omezit zvolením větší velikosti uzlu (stránky datového souboru). Do jednoho uzlu by se vešlo více záznamů a počet zřetězených uzlů by úměrně tomu poklesl. Další možností je použít na přetékající uzly jiný algoritmus než je lineární seznam. Pokud by délka řetězu uzlů překročila určitou hodnotu, použil by se první uzel pro uložení indexů zřetězených uzlů a mezi uzly by bylo možno vyhledávat půlením intervalů. Posledním zlepšením by mohlo být neplnit uzly do plné kapacity, ale ponechat v některých volné místo jako rezervu pro příští probublávání vkládaných klíčů. Případně do dlouhých řetězců vkládat za každý *k*-tý uzel jeden prázdný uzel navíc. Při vkládání by se probublávání zastavilo nejpozději po právě *k* uzlech.

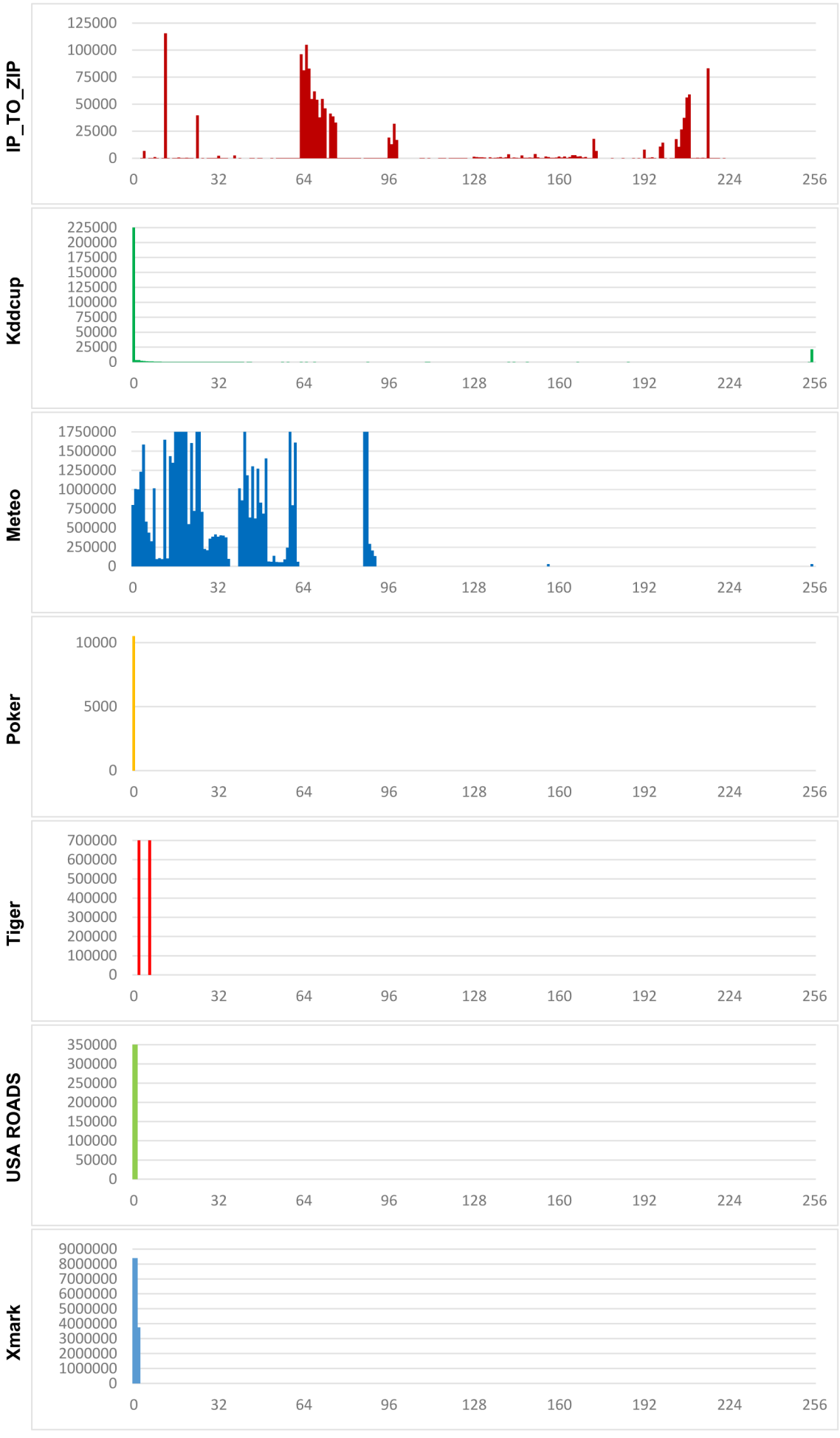
Změřený výkon hašovacích tabulek mě přesvědčil, že se jedná o datovou strukturu, kterou má smysl se zabývat. Stále však ještě zbývají určité aspekty které se musí dořešit před možností univerzálnějšího nasazení. Do té doby je hašovací tabulka se zachováním pořadí výborný nástroj, ale pouze pro specifické použití přizpůsobené zpracovávaným datům.

Použitá literatura

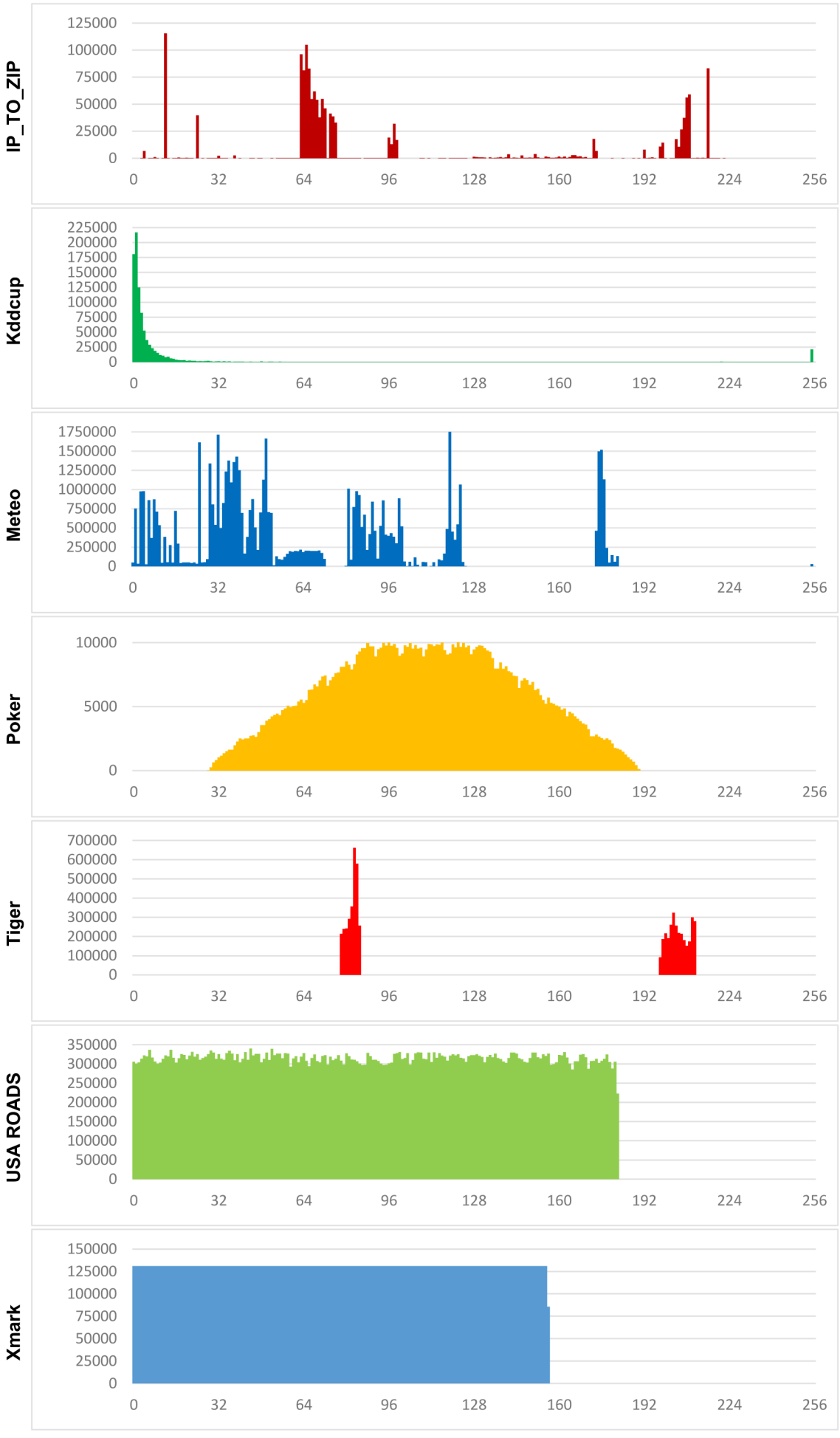
1. BRASS, Peter, *Advanced Data Structures*, Cambridge: Cambridge University Press, 2008.
2. FAGIN, R.; NIEVERGETL, J.; PIPPENGER, N.; STRONG, H. R., Extendible Hashing - A Fast Access Method for Dynamic Files, In *Transactions on Database Systems Journal*, New York, ACM, 1979, Volume 4 Issue 3, Sept., p. 315 – 344
3. LITWIN, Witold, Linear Hashing: A New Tool for File and Table Addressing, In *Proceedings of the Sixth International Conference on Very Large Data Bases*, Montreal, VLDB Endowment, 1980, Volume 6, p. 212 – 223
4. Extendible hashing – Wikipedia, [cit. 2015-03-19], < https://en.wikipedia.org/wiki/Extendible_hashing >
5. Hash table – Wikipedia, [cit. 2015-03-16], < https://en.wikipedia.org/wiki/Hash_table >
6. Hašovací tabulka – Wikipedie, [cit. 2015-03-15], < https://cs.wikipedia.org/wiki/Hašovací_tabulka >
7. Linear hashing – Wikipedia, [cit. 2015-03-18], < https://en.wikipedia.org/wiki/Linear_hashing >

Přílohy

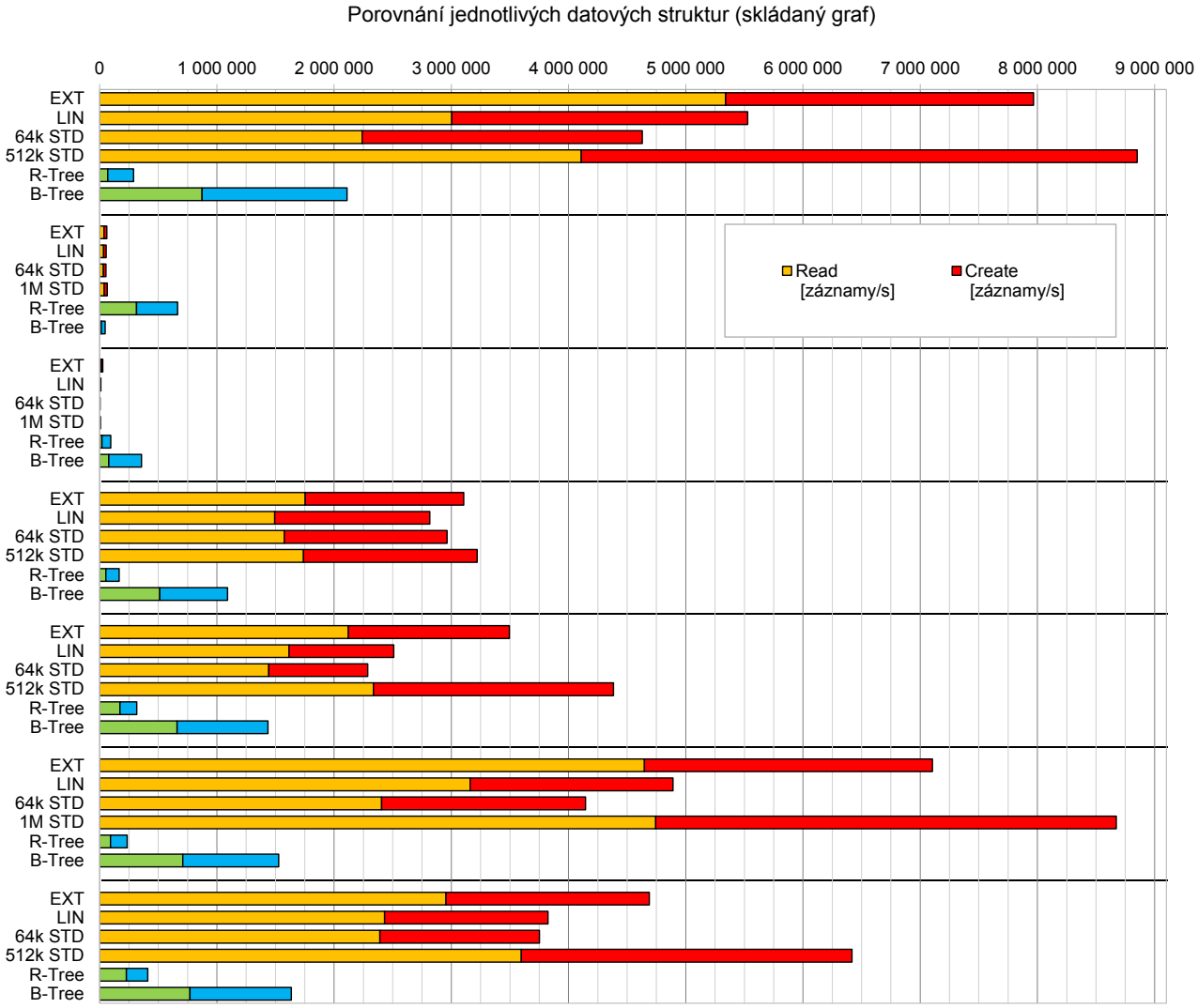
Příloha A1 - Výchozí distribuce otisků klíčů

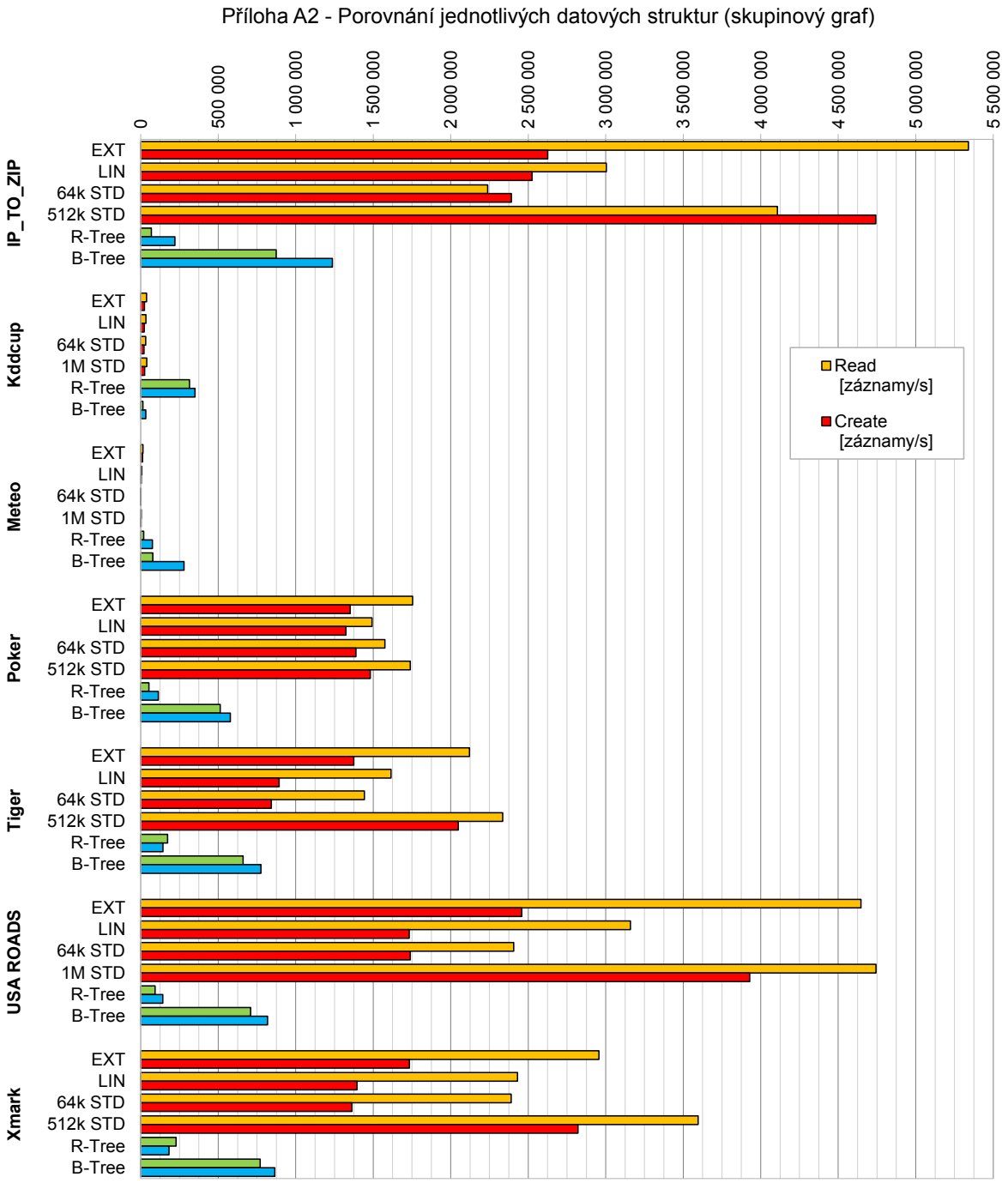


Příloha A1 - Optimalizovaná distribuce otisků klíčů

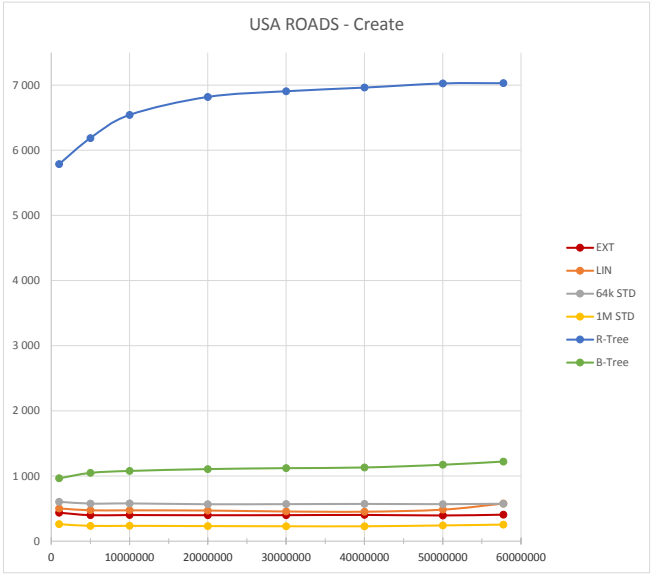


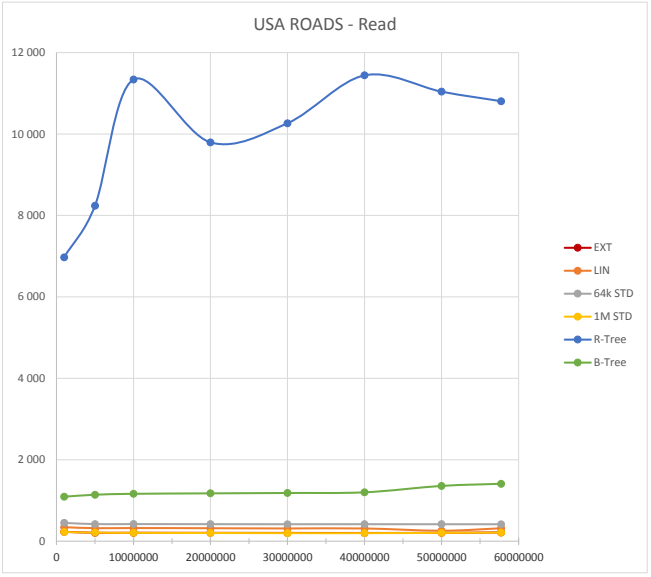
	Metoda	Nejdelší řetěz [uzly]	Create [záznamy/s]	Read [záznamy/s]
IP_TO_ZIP	EXT	4	2 626 262	5 340 067
	LIN	67	2 525 075	3 003 788
	64k STD	81	2 391 075	2 238 848
	512k STD	19	4 742 822	4 107 744
	R-Tree	4	220 495	68 648
	B-Tree	4	1 236 551	873 829
Kddcup	EXT	5498	24 402	37 067
	LIN	5498	21 953	33 818
	64k STD	5569	21 667	32 666
	1M STD	5498	25 378	38 566
	R-Tree	6	350 195	314 569
	B-Tree	10	32 682	14 010
Meteo	EXT	935	11 656	14 528
	LIN	1957	4 076	5 770
	64k STD	6565	1 027	1 277
	1M STD	0	2 401	2 960
	R-Tree	4	75 368	19 153
	B-Tree	4	279 222	77 742
Poker	EXT	1	1 351 351	1 754 386
	LIN	1	1 324 503	1 492 537
	64k STD	1	1 388 889	1 574 803
	512k STD	1	1 481 481	1 739 130
	R-Tree	4	113 572	52 784
	B-Tree	3	578 035	512 821
Tiger	EXT	2	1 374 703	2 120 758
	LIN	13	893 306	1 615 012
	64k STD	21	842 896	1 443 135
	512k STD	4	2 048 417	2 335 358
	R-Tree	3	143 267	173 239
	B-Tree	3	775 274	660 357
USA ROADS	EXT	1	2 458 000	4 646 933
	LIN	8	1 732 334	3 159 497
	64k STD	15	1 739 275	2 405 963
	1M STD	2	3 929 320	4 743 140
	R-Tree	4	142 245	92 542
	B-Tree	4	818 184	709 109
Xmark	EXT	1	1 734 043	2 955 636
	LIN	5	1 394 607	2 430 781
	64k STD	5	1 362 586	2 390 037
	512k STD	1	2 821 603	3 596 568
	R-Tree	3	182 431	227 945
	B-Tree	3	864 758	770 520



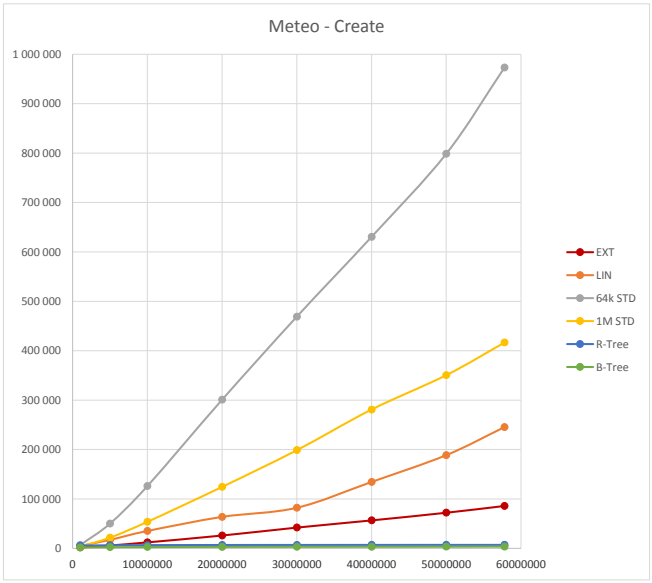

Příloha A2 - Porovnání jednotlivých datových struktur (skupinový graf)
0
500 000
1 000 000
1 500 000
2 000 000
2 500 000
3 000 000
3 500 000
4 000 000
4 500 000
5 000 000
5 500 000
IP_TO_ZIP
EXT
LIN
64k STD
512k STD
R-Tree
B-Tree
Kddcup
EXT
LIN
64k STD
1M STD
R-Tree
B-Tree
Meteo
EXT
LIN
64k STD
1M STD
R-Tree
B-Tree
Poker
EXT
LIN
64k STD
512k STD
R-Tree
B-Tree
Tiger
EXT
LIN
64k STD
512k STD
R-Tree
B-Tree
USA ROADS
EXT
LIN
64k STD
1M STD
R-Tree
B-Tree
Xmark
EXT
LIN
64k STD
512k STD
R-Tree
B-Tree
Read [záznamy/s]
Create [záznamy/s]

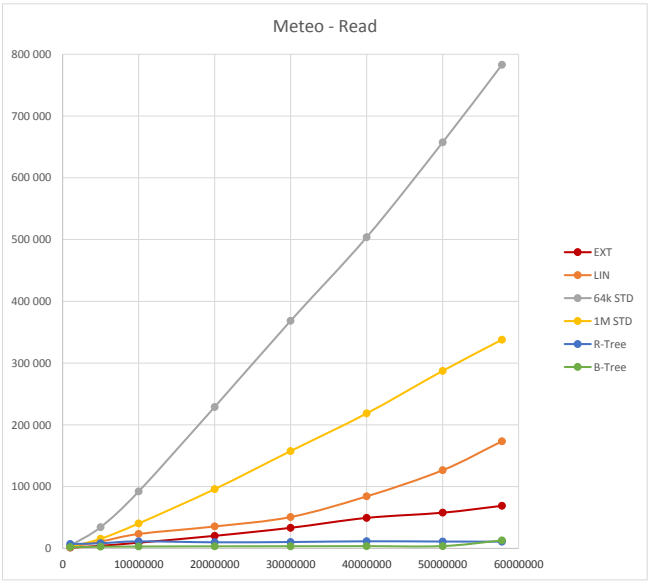
	Počet záznamů	USA ROADS - Čas potřebný pro zpracování 1 milionu záznamů EXT	LIN	64k STD	1M STD	R-Tree	B-Tree
Create	1000000	437	503	605	261	5 788	965
	5000000	399	474	577	235	6 188	1 048
	10000000	401	474	580	236	6 543	1 078
	20000000	398	470	567	231	6 819	1 107
	30000000	400	456	569	229	6 907	1 121
	40000000	403	452	572	229	6 963	1 131
	50000000	395	484	568	242	7 027	1 173
	57733497	407	577	575	254	7 030	1 222
Read	1000000	230	344	450	225	6 973	1 092
	5000000	208	319	420	210	8 239	1 140
	10000000	207	323	420	204	11 339	1 164
	20000000	205	318	418	201	9 795	1 175
	30000000	203	312	416	197	10 265	1 184
	40000000	202	312	417	197	11 442	1 200
	50000000	205	257	416	204	11 040	1 357
	57733497	215	317	416	211	10 806	1 410





	Počet záznamů	Meteo - Čas potřebný pro zpracování 1 milionu záznamů EXT	LIN	64k STD	1M STD	R-Tree	B-Tree
Create	1000000	1 796	4 891	6 529	3 314	5 788	1 802
	5000000	5 729	17 130	50 088	21 694	6 188	2 368
	10000000	12 073	35 155	125 925	53 619	6 543	2 671
	20000000	25 936	63 676	300 762	124 380	6 819	2 926
	30000000	41 998	82 153	468 882	198 646	6 907	3 091
	40000000	56 554	134 405	630 421	280 898	6 962	3 213
	50000000	72 075	188 651	798 631	350 493	7 027	3 318
	57796503	85 796	245 343	973 240	416 561	7 030	3 581
Read	1000000	1 068	2 285	4 400	2 348	7 019	2 020
	5000000	4 139	10 965	34 347	15 507	8 239	2 660
	10000000	9 180	23 269	92 069	40 282	11 339	2 949
	20000000	20 267	35 443	228 929	95 894	9 795	3 268
	30000000	33 250	50 721	368 503	157 552	10 265	3 387
	40000000	49 311	84 134	503 661	218 393	11 462	3 563
	50000000	57 731	126 484	657 346	287 315	11 040	3 683
	57796503	68 831	173 320	782 936	337 792	10 806	12 863





Příloha B – Disk DVD

Součástí bakalářské práce je DVD.

Adresářová struktura přiloženého DVD:

`\code`	Zdrojové kódy
`\code\test\paged\hashtable_test`	Zdrojový kód testovací aplikace + projekt pro Visual Studio 2013
`\code\dstruct\paged\hashtable`	Zdrojové kódy implementovaných hašovacích algoritmů
`\text`	Text bakalářské práce a přílohy v pdf + výsledky měření v souboru xls
`\collection`	Testované datové kolekce